FLORIN LEON

APLICAȚII DE INGINERIA PROGRAMĂRII ÎN C#

editura Tehnopress

FLORIN LEON

APLICAȚII DE INGINERIA PROGRAMĂRII ÎN C#

editura
Tehnopress
IAȘI – 2012

Referenţi ştiinţifici:
Prof. dr. ing. **Silvia CURTEANU**
Prof. dr. ing. **Doru PĂNESCU**

Editura TEHNOPRESS
Str. Pinului nr. 1A
700109 Iaşi
Tel./fax: 0232 260092
E-mail: tehnopress@yahoo.com
http://www.tehnopress.ro
Editură acreditată CNCSIS, cod CNCSIS 89

Descrierea CIP a Bibliotecii Naţionale a României
LEON, FLORIN
Aplicaţii de ingineria programării în C# / Florin Leon.
– Iaşi: Tehnopress, 2012
Bibliogr.
ISBN 978-973-702-909-6

004.43 C#

Cuprins

Capitolul 1. Compilarea, decompilarea şi obscurizarea programelor C#

1. Obiective 9
2. Structura generală a unui program C# 10
   2.1. Clase 10
   2.2. Clase statice şi membri statici 11
   2.3. Câmpuri constante 13
   2.4. Structuri 14
   2.5. Trimiterea argumentelor prin referinţă 16
   2.6. Operaţiile de *boxing* şi *unboxing* 16
   2.7. Enumeraţii 17
3. Compilarea, decompilarea şi obscurizarea codului 18
   3.1. Compilarea. Limbajul Intermediar Comun 18
   3.2. Decompilarea. Programul *.NET Reflector* 19
   3.3. Compilarea în modurile *Debug* şi *Release* 23
   3.4. Obscurizarea codului. *Dotfuscator* 29
4. Aplicaţii 30

Capitolul 2. Stilul de scriere a codului. Tratarea excepţiilor. Interfaţa grafică cu utilizatorul

1. Obiective 35
2. Stilul de scriere a codului 35
   2.1. Acoladele 36
   2.2. Standarde de programare 37
   2.3. Convenţii pentru nume 39
3. Tratarea excepţiilor 42
   3.1. Tratarea excepţiilor pe firul de execuţie al aplicaţiei 46
4. Interfaţa cu utilizatorul 47
   4.1. Proiectarea comenzilor şi interacţiunilor 47
   4.2. Considerente practice 48
   4.3. Profilurile utilizatorilor 50
5. Interfaţa grafică cu utilizatorul în Microsoft Visual Studio .NET 51
6. Elemente de C# 58
   6.1. Clase parţiale 58
   6.2. Proprietăţi. Accesori 59
      6.2.1. Accesorul *get* 60
      6.2.2. Accesorul *set* 61

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

6.2.3. Aspecte mai complexe ale lucrului cu proprietăţi....62
7. Aplicaţii....65

Capitolul 3. Reutilizarea codului cu ajutorul DLL-urilor

1. Obiective....69
2. Bibliotecile legate dinamic....69
3. Crearea DLL-urilor în C#....70
3.1. Legarea statică....72
3.2. Legarea dinamică....72
3.3. Depanarea unui DLL....74
4. Grafică în C#....75
5. Aplicaţii....77

Capitolul 4. Documentarea unui proiect. Fişiere de ajutor

1. Obiective....81
2. Crearea de fişiere de ajutor....81
2.1. Crearea de fişiere HLP....81
2.2. Crearea de fişiere CHM....86
3. Activarea unui fişier de ajutor prin program....88
3.1. *Process.Start*....88
3.2. *HelpProvider*....88
3.3. *Help*....89
4. Generarea automată a documentaţiei *API*....90
5. Comentariile....92
6. Lucrul cu fişiere în C#: încărcare, salvare....95
7. Aplicaţii....96

Capitolul 5. Diagrame UML

1. Obiective....105
2. Diagrame principale ale UML....105
2.1. Diagrama cazurilor de utilizare....105
2.2. Diagrama de clase....107
2.2.1. Dependenţa....107
2.2.2. Asocierea....108
2.2.3. Agregarea şi compunerea....110
2.2.4. Moştenirea....110
2.2.5. Metode abstracte şi virtuale....112
2.2.6. Interfeţe....113
2.2.7. Trăsături statice....113
2.3. Diagrame de activităţi....114
2.4. Diagrama de secvenţe....117
3. *Altova UModel*....118

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

3.1. Diagrama de clase ... 119
3.1.1. Generarea de cod ... 120
3.1.2. Crearea diagramei unui proiect existent ... 125
3.1.3. Aranjarea automată a elementelor din diagrame ... 128
3.2. Celelalte diagrame ... 128
4. Aplicaţii ... 128

Capitolul 6. Arhitectura MVC

1. Obiective ... 131
2. Introducere. Arhitectura cu trei straturi ... 131
3. Arhitectura *MVC* ... 133
4. Arhitectura *MVP* ... 135
4.1. Variante de actualizare a *Vizualizării* ... 136
5. Aplicaţii ... 138

Capitolul 7. Şablonul de proiectare Metoda Fabrică

1. Obiective ... 151
2. Şablonul creaţional *Metoda Fabrică* ... 151
3. Exemplu de implementare ... 152
4. Moştenirea şi polimorfismul ... 154
4.1. Polimorfismul ... 154
4.2. Clase abstracte ... 154
4.3. Interfeţe ... 155
4.4. Membri virtuali ... 156
4.5. Clase sigilate şi membri sigilaţi ... 158
4.6. Înlocuirea unui membru cu ajutorul cuvântului cheie *new* ... 159
4.7. Accesarea clasei de bază cu ajutorul cuvântului cheie *base* ... 160
5. Aplicaţii ... 161

Capitolul 8. Şabloanele de proiectare Singleton şi Prototip

1. Obiective ... 167
2. Şablonul creaţional *Singleton* ... 167
2.1. Exemplu de implementare ... 168
3. Şablonul creaţional *Prototip* ... 169
3.1. Exemplu de implementare ... 170
4. Aplicaţii ... 172

Capitolul 9. Şablonul de proiectare Faţadă

1. Obiectiv ... 185
2. Scop şi motivaţie ... 185
3. Aplicabilitate ... 186
4. Analiza şablonului ... 187

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

5. Exemplu de implementare 188
6. Aplicaţie 190

Capitolul 10. Şablonul de proiectare Proxy

1. Obiectiv 193
2. Scop şi motivaţie 193
3. Aplicabilitate 194
4. Analiza şablonului 195
5. Exemplu de implementare 196
6. Aplicaţii 197

Capitolul 11. Şablonul de proiectare Comandă

1. Obiective 211
2. Scop şi motivaţie 211
3. Aplicabilitate 214
4. Analiza şablonului 215
5. Exemplu de implementare 216
6. Aplicaţie 218

Capitolul 12. Evaluarea vitezei de execuţie a unui program

1. Obiective 231
2. Metoda *DateTime* 231
3. Pointeri în C# 232
4. Metoda *PerformanceCounter* 234
   4.1. Metode de accelerare 235
5. Metoda *Stopwatch* 236
6. Compilarea *JIT* 236
7. Aplicaţii 238

Capitolul 13. Testarea unităţilor cu NUnit

1. Obiectiv 243
2. Testarea unităţilor 243
3. Utilizarea platformei *NUnit* 244
4. Aplicaţii 249

Capitolul 14. Rapoarte de testare

1. Obiective 253
2. Testarea unei ierarhii de clase 253
3. Aplicaţie 256

Referinţe 263

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Cuvânt înainte

În anul 2001 am început să predau laboratoarele de *Ingineria programării* la Facultatea de Automatică şi Calculatoare de la Universitatea Tehnică „Gheorghe Asachi" din Iaşi. La acel moment, în unele facultăți de profil din România, aceste laboratoare se concentrau pe programare vizuală Windows, astfel încât în primii ani conținutul s-a axat pe programare folosind *Borland C++ Builder* şi *Microsoft Visual Studio* cu *Microsoft Foundation Classes (MFC)*. Pe lângă aspectele de programare vizuală, am prezentat încă de pe atunci chestiuni legate de modularizare, utilizarea DLL-urilor, crearea aplicațiilor COM, tratarea excepțiilor, realizarea de fişiere de ajutor (*help*), diagrame UML şi evaluarea vitezei de execuție a programelor.

În anul 2003 am introdus limbajul *C#* pentru partea de programare a laboratoarelor, limbaj de care m-am ataşat încă de la apariția sa, în 2002, şi pe care am continuat să îl utilizez de atunci pentru toate proiectele de programare de natură profesională sau personală.

Din 2008, odată cu rescrierea cursurilor de *Ingineria programării*, am modificat şi laboratoarele, adăugând o parte substanțială legată de proiectarea şi testarea aplicațiilor. Astfel, un laborator tratează şabloane arhitecturale, cinci se referă la şabloane de proiectare şi două la testarea unităților. Laboratoarele, în forma lor actuală, constituie rezultatul experienței acumulate în timp, încercând să ofer o viziune practică asupra problemelor complexe legate de realizarea produselor software comerciale.

Sunt recunoscător tuturor studenților pentru semnalarea neclarităților şi erorilor strecurate în laboratoare. De asemenea, mulțumesc studenților Veridiana Mărtişcă, Simona Scripcaru, Liudmila Tofan, Florin Alexandru Hodorogea şi Alexandru Gologan pentru observațiile asupra versiunii preliminare a acestui ghid.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 1

Compilarea, decompilarea şi obscurizarea programelor C#

1. Obiective
2. Structura generală a unui program C#
3. Compilarea, decompilarea şi obscurizarea codului
4. Aplicații

1. Obiective

Aplicațiile de ingineria programării nu se doresc a fi, în primul rând, aplicații de programare în C#. Din păcate, programe complexe la standarde comerciale nu se pot termina în două ore, deci problemele vor avea o natură academică surprinzând însă chestiuni ce se pot regăsi în aplicațiile din industrie şi care trebuie rezolvate în principal la standarde înalte de calitate.

Accentul principal al prezentului ghid cade pe proiectarea programelor, folosind de exemplu şabloane de proiectare, pe modul cum se gândeşte şi se scrie un program, pe testare şi pe crearea diverselor tipuri de documente aferente.

Vom utiliza limbajul C# pentru că este un limbaj modern, special destinat dezvoltării rapide de aplicații. Când vom considera necesar, vom prezenta şi noțiuni de programare în C#, mai ales în primele trei capitole. Ca mediu de dezvoltare, vom întrebuința Microsoft Visual Studio 2005.

Obiectivele primului capitol sunt următoarele:

1. Prezentarea modului de organizare a unui program C# care conține clase, structuri şi enumerații. Discutarea diferențelor dintre tipurile referință (clase) şi tipurile valoare (structuri);
2. Precizarea diferențelor de compilare în modurile *Debug* şi *Release*;
3. Descrierea posibilităților de decompilare a aplicațiilor .NET şi de protejare a acestora prin obscurizarea codului.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

2. Structura generală a unui program C#

O soluție C# constă dintr-unul sau mai multe proiecte. Proiectele constau dintr-unul sau mai multe fișiere. Fișierele pot conține zero sau mai multe spații de nume (engl. "namespaces"). Un *namespace* poate conține tipuri precum clase, structuri, enumerații, dar și alte *namespace*-uri. Mai jos este prezentat un schelet al unui program C# alcătuit din aceste elemente.

```
using System;

namespace MyNamespace
{
   class MyClass
   {
   }

   struct MyStruct
   {
   }

   enum MyEnum
   {
   }

   class MyMainClass
   {
      static void Main(string[] args)
      {
         // începutul programului propriu-zis
      }
   }
}
```

2.1. Clase

Clasa este cel mai important tip de date în C#. În următorul exemplu, se definește o clasă publică având un câmp, o metodă și un constructor. De remarcat terminologia utilizată pentru o variabilă membru, *câmp*, deoarece termenul *proprietate* reprezintă un alt concept C# pe care îl vom discuta în capitolul 2.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

În exemplul de mai jos, problema este definirea unui câmp public. Conform teoriei programării orientate obiect, toate câmpurile trebuie să fie private iar accesul la ele să se facă prin metode publice.

```
public class Person
{
   // Câmp / Field
   public string name;       // !!! câmp public, nerecomandat

   // Constructor
   public Person()
   {
      name = "nedefinit";
   }

   // Metodă / Method
   public void ChangeName(string newName)
   {
      name = newName;
   }
}

class TestPerson
{
   static void Main()
   {
      Person person1 = new Person();
      Console.WriteLine(person1.name);

      person1.ChangeName("Ion Popescu");
      Console.WriteLine(person1.name);
   }
}
```

2.2. Clase statice şi membri statici

O clasă statică nu poate fi instanțiată. Deoarece nu există instanțe ale clasei, apelarea unei metode dintr-o clasă statică se realizează folosind numele clasei înseşi. De exemplu, dacă avem o clasă statică numită UtilityClass care conține o metodă publică numită MethodA, aceasta este apelată în modul următor:

```
UtilityClass.MethodA();
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

O clasă statică poate fi utilizată ca o modalitate convenabilă de a grupa o serie de metode care operează asupra unor parametri de intrare şi nu au nevoie de câmpuri întrucât nu au stare internă. De exemplu, în mediul .NET, clasa statică System.Math conține metode care realizează operații matematice, fără a avea nevoie să memoreze sau să acceseze alte date în afara argumentelor cu care sunt apelate.

Mai jos este prezentat un exemplu de clasă statică având două metode care convertesc temperatura din grade Celsius în grade Fahrenheit şi viceversa.

```
public static class TemperatureConverter
{
   public static double CelsiusToFahrenheit(string temperatureCelsius)
   {
      // conversia argumentului din string în double
      double celsius = Convert.ToDouble(temperatureCelsius);

      // conversia din grade Celsius în grade Fahrenheit
      double fahrenheit = (celsius * 9 / 5) + 32;

      return fahrenheit;
   }

   public static double FahrenheitToCelsius(string temperatureFahrenheit)
   {
      double fahrenheit = Convert.ToDouble(temperatureFahrenheit);
      double celsius = (fahrenheit – 32) * 5 / 9;
      return celsius;
   }
}

class TestTemperatureConverter
{
   static void Main()
   {
      Console.WriteLine("Selectati directia de conversie");
      Console.WriteLine("1. Din grade Celsius in grade Fahrenheit");
      Console.WriteLine("2. Din grade Fahrenheit in grade Celsius");
      Console.Write(":");
      string selection = Console.ReadLine();
      double f, c = 0;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
      switch (selection)
      {
         case "1":
            Console.Write("Introduceti temperatura in grade Celsius: ");
            f = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
            // se afişează rezultatul cu 2 zecimale
            Console.WriteLine("Temperatura in grade Fahrenheit: {0:F2}", f);
            break;

         case "2":
            Console.Write("Introduceti temperatura in grade Fahrenheit: ");
            c = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
            Console.WriteLine("Temperatura in grade Celsius: {0:F2}", c);
            break;

         default:
            Console.WriteLine("Selectati un tip de conversie");
            break;
      }

      // aşteaptă apăsarea tastei ENTER
      Console.ReadLine();
   }
}
```

Deseori, se utilizează *clase nestatice cu membri statici* în locul claselor statice. În acest caz, membrii statici pot fi apelați chiar şi înaintea creării unor instanțe ale clasei. La fel ca mai sus, membrii statici sunt accesați cu numele clasei, nu al unei instanțe. Indiferent câte instanțe ale clasei sunt create, pentru un membru static există întotdeauna o singură copie. Metodele statice nu pot accesa metode şi câmpuri nestatice din clasă.

Câmpurile statice se folosesc în general pentru a stoca valori care trebuie cunoscute de către toate instanțele clasei şi pentru a păstra evidența numărului de obiecte care au fost instanțiate.

2.3. Câmpuri constante

Un câmp constant este static în comportament (nu poate fi modificat) şi de aceea aparține tot tipului şi nu instanțelor. Prin urmare va fi accesat tot cu numele clasei, ca şi câmpurile statice.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
public class Car
{
   public const int NumberOfWheels = 4;
   public static void Drive() { }

   // alte câmpuri şi metode nestatice
}


// utilizare din exteriorul clasei
Car.Drive();
int i = Car.NumberOfWheels;
```

2.4. Structuri

În C#, structurile sunt versiuni simplificate ale claselor. De obicei, ele ocupă mai puțin spațiu în memorie şi sunt potrivite pentru tipurile de date de dimensiuni mici, utilizate frecvent. Diferența cea mai importantă între structuri şi clase este faptul că structurile sunt *tipuri valoare* iar clasele sunt *tipuri referință*.

Când se creează o instanță de tip valoare, se alocă în memoria *stivă* un singur spațiu pentru păstrarea valorii instanței respective. În acest mod sunt tratate tipurile primitive precum int, float, bool, char etc. Compilatorul creează automat un constructor implicit care inițializează toate câmpurile cu valorile implicite ale tipurilor acestora, de exemplu tipurile numerice cu 0, bool cu false, char cu '\0' iar câmpurile de tip referință (instanțe ale altor clase) cu null. Pentru structuri nu se poate declara un constructor implicit (fără parametri), însă se pot declara constructori cu parametri, care să inițializeze membrii cu valori diferite de cele implicite. Dezalocarea instanțelor se face automat când acestea ies din domeniul lor de definiție.

La alocarea instanțelor de tip referință, se memorează atât referința obiectului în *stivă*, cât şi spațiul pentru conținutul obiectului în *heap*. Managementul memoriei este făcut de către *garbage collector*.

Să considerăm următoarea situație: structura MyPoint şi clasa MyForm.

```
MyPoint p1;          // p1 este alocat cu valorile implicite ale membrilor
p1 = new MyPoint();  // nu are efect aici, resetează valorile membrilor
MyForm f1;           // se alocă referința, f1 = null
f1 = new MyForm();   // se alocă obiectul, f1 primeşte referința acestuia
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

În primul caz, se alocă un singur spațiu în memorie pentru p1. În al doilea caz, se alocă două spații: unul pentru obiectul MyForm şi unul pentru referința lui, f1. De aceea, dacă vrem să declarăm un vector de 1000 de puncte, este mai avantajos să creăm o structură decât o clasă, deoarece astfel vom aloca mai puțină memorie.

Dacă vrem să copiem obiectele:

```
MyPoint p2 = p1;
MyForm f2 = f1;
```

p2 devine o copie independentă a lui p1, fiecare cu câmpurile lui separate. În cazul lui f2, se copiază numai referința, astfel încât f1 şi f2 pointează către acelaşi obiect.

Fie metoda următoare, apelată cu argumentele p1 şi f1: Change(p1, f1):

```
void Change(MyPoint p, MyForm f)
{
   p.X = 10;          // p este o copie, instrucţiunea nu are efect asupra lui p1
   f.Text = "Hello"; // f şi f1 pointează la acelaşi obiect, f1.Text se schimbă
   f = null;          // f este o copie a referinţei f1, instrucţiunea nu are efect asupra lui f1
}
```

Pentru o compatibilitate cât mai bună cu mediul .NET, *Biblioteca MSDN* recomandă utilizarea structurilor numai în următoarele situații:

- Tipul reprezintă o valoare unitară, similară cu un tip primitiv (int, double etc.);
- Dimensiunea unei instanțe este mai mică de 16 octeți (deoarece la transmiterea ca parametru în metode se creează o nouă copie pe stivă);
- Tipul este *immutable* (metodele nu modifică valorile câmpurilor; când se doreşte schimbarea acestora se creează un nou obiect cu noile valori);
- Operațiile de *boxing* (împachetare) şi *unboxing* (despachetare), prezentate în secțiunea 2.6, sunt rare.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

2.5. Trimiterea argumentelor prin referinţă

Trimiterea argumentelor prin referință se realizează cu ajutorul cuvintelor cheie ref şi out. Astfel, modificările făcute asupra parametrului în metoda apelată se vor reflecta asupra variabilei din metoda apelantă. Un argument trimis ca ref trebuie inițializat mai întâi. Un argument trimis cu out nu trebuie inițializat în metoda apelantă, însă metoda apelată este obligată să îi atribuie o valoare.

```
class RefExample
{
   static void Method(ref int i)
   {
      i = 44;
   }
   static void Main()
   {
      int val = 0;
      Method(ref val);
      // val este acum 44
   }
}
```

```
class OutExample
{
   static void Method(out int i)
   {
      i = 44;
   }
   static void Main()
   {
      int val;
      Method(out val);
      // val este acum 44
   }
}
```

Revenind la exemplul cu structura MyPoint şi clasa MyForm, fie metoda următoare, apelată cu argumentele p1 şi f1: Change(ref p1, ref f1):

```
void Change(ref MyPoint p, ref MyForm f)
{
   p.X = 10;              // se modifică p1.X
   f.Text = "Hello";   // se modifică f1.Text
   f = null;              // f1 este distrus
}
```

2.6. Operaţiile de *boxing* şi *unboxing*

Aceste operații permit ca tipurile valoare să fie tratate drept tipuri referință, după cum se poate vedea în figura 1.1.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

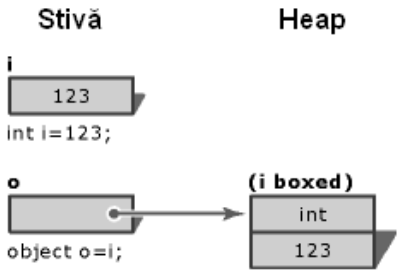
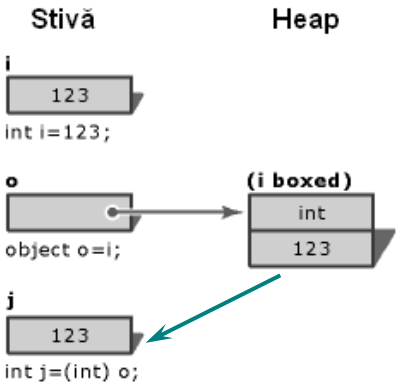
Boxing (împachetare)	*Unboxing (despachetare)*
int i = 123; *// tip valoare* **object** o = i; *// tip referinţă (boxing)*	**int** i = 123; **object** o = i; **int** j = (**int**)o; *// tip valoare (unboxing)*
	

Figura 1.1. *Operaţiile de boxing şi unboxing*

Vom utiliza aceste operații în capitolul 3.

2.7. Enumeraţii

O enumerație este alcătuită dintr-o mulțime de constante. Enumerațiile pot avea orice tip integral cu excepția lui char, tipul implicit fiind int. Valoarea implicită a primului element este 0, iar valorile succesive sunt incrementate cu 1. De exemplu:

```
enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
```

În această enumerație, Sun este 0, Mon este 1, Tue este 2 şi aşa mai departe. Enumeratorii pot avea inițializatori care suprascriu valorile implicite. De exemplu:

```
enum Zile { Lun = 1, Mar, Mie, Joi, Vin, Sam, Dum };
```

Aici secvența de elemente pornește de la 1 în loc de 0.
Următoarele instrucțiuni sunt valide:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
int x = (int)Zile.Lun;          // x = 1
Zile z1 = Zile.Mar;             // z1 = Mar
Zile z2 = (Zile)3;              // z2 = Mie
string s = z2.ToString();       // s = "Mie"
```

Modificarea valorilor unei enumerații într-o nouă versiune a unui program poate cauza probleme pentru alte programe ce folosesc codul respectiv. De multe ori valorile din enum sunt utilizate în instrucțiuni switch, iar dacă noi elemente sunt adăugate enumerației, se va activa cazul default. Dacă alți dezvoltatori depind de acel cod, ei trebuie să știe cum să trateze noile elemente adăugate.

3. Compilarea, decompilarea și obscurizarea codului

3.1. Compilarea. Limbajul Intermediar Comun

Limbajul Intermediar Comun (engl. "Common Intermediate Language", CIL), cunoscut și sub denumirea de *Limbajul Intermediar Microsoft* (engl. "Microsoft Intermediate Language", MSIL) este limbajul de nivelul cel mai scăzut al platformei .NET. MSIL a fost numele utilizat pentru limbajul intermediar până la versiunea 1.1 a platformei .NET. Începând cu versiunea 2.0, limbajul a fost standardizat iar denumirea standardului este CIL.

Compilarea și execuția unui program .NET se realizează în două etape, după cum se prezintă în figura 1.2.

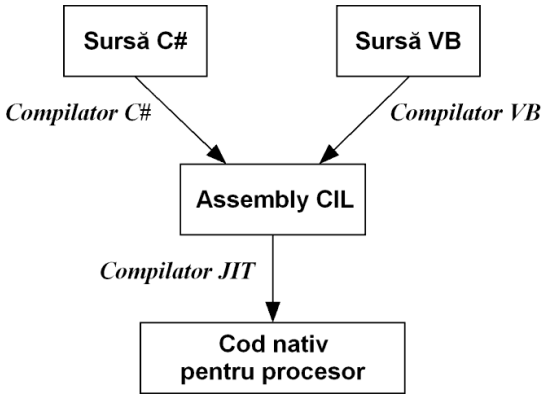


Figura 1.2. *Etapele compilării și execuției unui program .NET*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

În timpul compilării limbajelor .NET, codul sursă este transformat în cod CIL şi nu direct în cod executabil de către procesor. CIL reprezintă un set de instrucțiuni independent de sistemul de operare şi de procesor, care poate fi executat în orice mediu pe care este instalată platforma .NET, de exemplu motorul de execuție (*runtime*-ul) .NET pentru Windows, sau *Mono* pentru Linux.

Compilarea „la timp" (engl. "just-in-time compilation", JIT) are loc în momentul execuției efective a programului şi presupune transformarea codului CIL în instrucțiuni executabile imediat de către procesor. Conversia se realizează gradat în timpul execuției programului, iar compilatorul *JIT* efectuează o serie de optimizări specifice mediului de execuție.

Avantajul principal al platformei .NET este interoperabilitatea dintre diferite limbaje de programare. De exemplu, un proiect scris în Visual Basic poate fi apelat fără modificări dintr-un proiect C#.

3.2. Decompilarea. Programul *.NET Reflector*

Deoarece codul intermediar este standardizat, este relativ simplă transformarea inversă, într-un limbaj de nivel înalt precum C#. Un astfel de decompilator este *.NET Reflector*, pe care *MSDN Magazine* l-a numit unul din utilitarele obligatorii pentru un dezvoltator .NET. Programul este folosit deseori de către programatori pentru a înțelege structura internă a bibliotecilor .NET pentru care codul sursă nu este disponibil. Să considerăm următorul program simplu:

```
public class Program
{
   static void Main(string[] args)
   {
      string[] s = new string[] { "Hello, ", "World!" };
      for (int i = 0; i < s.Length; i++)
         Console.Write(s[i]);
      Console.WriteLine(Environment.NewLine + "ok");  // NewLine pentru Windows este "\r\n"
   }
}
```

După compilare, *assembly*-ul rezultat (în acest caz fişierul *exe*) se deschide în *.NET Reflector*. Programul permite navigarea prin *namespace*-uri, clase şi metode. Cu click-dreapta se deschide un meniu din care se poate selecta opțiunea de decompilare (engl. "disassemble"), ca în

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

figura 1.3. Din *combo-box*-ul din bara de instrumente se alege limbajul în care să se realizeze decompilarea.

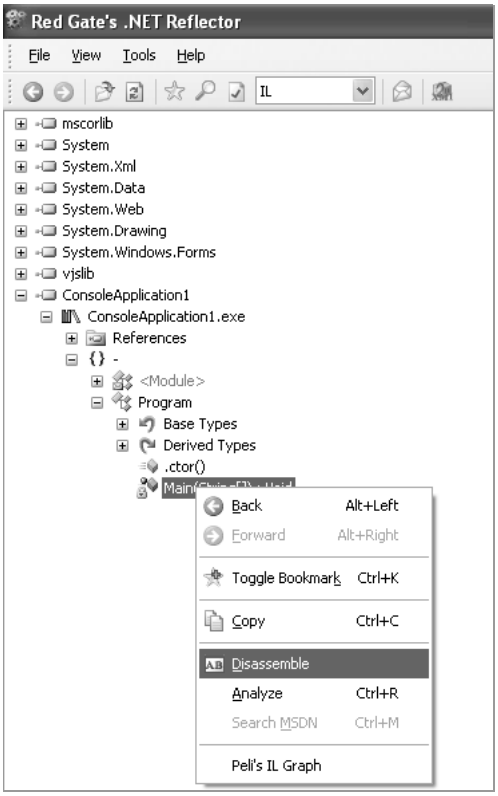


Figura 1.3. *Programul .NET Reflector*

Iată rezultatele decompilărilor în mai multe limbaje:

C#

```
private static void Main(string[] args)
{
   string[] s = new string[] { "Hello, ", "World!" };
   for (int i = 0; i < s.Length; i++)
   {
      Console.Write(s[i]);
   }
   Console.WriteLine(Environment.NewLine + "ok");
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Visual Basic

```
Private Shared Sub Main(ByVal args As String())
   Dim s As String() = New String() { "Hello, ", "World!" }
   Dim i As Integer
   For i = 0 To s.Length - 1
      Console.Write(s(i))
   Next i
   Console.WriteLine((Environment.NewLine & "ok"))
End Sub
```

Managed C++

```
private: static void __gc* Main(String __gc* args __gc [])
{
   String __gc* s __gc [] = __gc new String __gc*[2] {
      S"Hello, ", S"World!"};
   for (Int32 __gc* i = 0; (i < s->Length); i++)
   {
      Console::Write(s[i]);
   }
   Console::WriteLine(String::Concat(Environment::NewLine, S"ok"));
}
```

Delphi

```
procedure Program.Main(args: string[]);
begin
   s := New(array[2] of string, ( ( 'Hello, ', 'World!' ) ));
   i := 0;
   while ((i < s.Length)) do
   begin
      Console.Write(s[i]);
      inc(i)
   end;
   Console.WriteLine(string.Concat(Environment.NewLine, 'ok'))
end;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

CIL

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  .maxstack 3
  .locals init (
    [0] string[] s,
    [1] int32 i,
    [2] string[] CS$0$0000,
    [3] bool CS$4$0001)
  L_0000: nop
  L_0001: ldc.i4.2
  L_0002: newarr string
  L_0007: stloc.2
  L_0008: ldloc.2
  L_0009: ldc.i4.0
  L_000a: ldstr "Hello, "
  L_000f: stelem.ref
  L_0010: ldloc.2
  L_0011: ldc.i4.1
  L_0012: ldstr "World!"
  L_0017: stelem.ref
  L_0018: ldloc.2
  L_0019: stloc.0
  L_001a: ldc.i4.0
  L_001b: stloc.1
  L_001c: br.s L_002b
  L_001e: ldloc.0
  L_001f: ldloc.1
  L_0020: ldelem.ref
  L_0021: call void [mscorlib]System.Console::Write(string)
  L_0026: nop
  L_0027: ldloc.1
  L_0028: ldc.i4.1
  L_0029: add
  L_002a: stloc.1
  L_002b: ldloc.1
  L_002c: ldloc.0
  L_002d: ldlen
  L_002e: conv.i4
  L_002f: clt
  L_0031: stloc.3
  L_0032: ldloc.3
  L_0033: brtrue.s L_001e
  L_0035: call string [mscorlib]System.Environment::get_NewLine()
  L_003a: ldstr "ok"
  L_003f: call string [mscorlib]System.String::Concat(string, string)
  L_0044: call void [mscorlib]System.Console::WriteLine(string)
  L_0049: nop
  L_004a: ret
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

3.3. Compilarea în modurile *Debug* şi *Release*

Deşi majoritatea optimizărilor se realizează în momentul compilării *JIT*, chiar şi compilatorul C# poate efectua analize ale codului şi unele simplificări în vederea creşterii vitezei de execuţie. Compilarea în mod *Debug* este destinată facilitării procesului de descoperire a erorilor şi de aceea codul generat urmează mai fidel structura codului sursă. În modul *Debug*, compilatorul *JIT* generează un cod mai uşor de depanat, însă mai lent.

În schimb, compilarea în mod *Release* poate introduce optimizări suplimentare. Aceste opţiuni pot fi controlate din mediul Visual Studio, astfel: *View → Solution Explorer → Project Properties → Build*. În modul *Release*, opţiunea *Optimize code* este activată.

De asemenea, în *View → Solution Explorer → Project Properties → Build → Advanced → Output*, se precizează crearea sau nu a unui fişier *pdb* (*program database*) care conţine informaţii ce fac legătura între codul CIL generat şi codul sursă iniţial, utile în special în faza de *Debug*.

În continuare, vor fi prezentate unele diferenţe de compilare în mod *Debug* (prima imagine, de sus), respectiv *Release* (a doua imagine, de jos).

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

a) Declararea variabilelor în locul în care sunt utilizate.

```
public Chromosome(GAParameters parameters, double max)
{
    int i;
    this.NoGenes = parameters.EncodingInfo.NoGenes;
    this.RealGenes = new double[this.NoGenes];
    for (i = 0; i < 3; i++)
    {
        this.RealGenes[i] = parameters.EncodingInfo.MinValu
    }
    for (i = 3; i < this.NoGenes; i++)
    {
        this.RealGenes[i] = ((ExtendedRandom.NextUniform()
    }
}
```

```
public Chromosome(GAParameters parameters, double max)
{

    this.NoGenes = parameters.EncodingInfo.NoGenes;
    this.RealGenes = new double[this.NoGenes];
    for (int i = 0; i < 3; i++)
    {
        this.RealGenes[i] = parameters.EncodingInfo.MinValu
    }
    for (int i = 3; i < this.NoGenes; i++)
    {
        this.RealGenes[i] = ((ExtendedRandom.NextUniform()
    }
}
```

Interesant este faptul că aceste optimizări nu apar întotdeauna. De exemplu, o metodă simplă cum ar fi următoarea nu va fi optimizată, deşi principiul este acelaşi ca mai sus.

```
public void Locals()
{
  int i;
  for (i = 0; i < 3; i++)
    DoSomething();
  for (i = 2; i < 5; i++)
    DoSomething();
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

b) Transformarea buclelor while în bucle for

```
int generations = 0;
int i = 0;
while (i < this._problem.Parameters.PopulationSize)
{
    this._population[i] = new Chromosome(this._problem.Parameters
    this._population[i].ComputeFitness(this._problem);
    i++;
}
base.Solution = Selection.GetBest(this._population);
Settings.StopTraining = false;
while (!searchOver)
{
    i = 0;
    while (i < this._problem.Parameters.PopulationSize)
    {
        int d1;
        int d2;
        int m;
        this.GetCandidates(i, this._problem.Parameters.Population
        Chromosome diffVect = new Chromosome(this._population[d1]
        Chromosome trialVect = new Chromosome(this._population[m]
```

```
int generations = 0;
for (int i = 0; i < this._problem.Parameters.PopulationSize; i++)

{
    this._population[i] = new Chromosome(this._problem.Parameters
    this._population[i].ComputeFitness(this._problem);

}
base.Solution = Selection.GetBest(this._population);
Settings.StopTraining = false;
while (!searchOver)
{
    for (int i = 0; i < this._problem.Parameters.PopulationSize;

    {
        int d1;
        int d2;
        int m;
        this.GetCandidates(i, this._problem.Parameters.Population
        Chromosome diffVect = new Chromosome(this._population[d1]
        Chromosome trialVect = new Chromosome(this._population[m]
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

c) Eliminarea inițializărilor cu null

```
internal class EvolutionaryHillClimbing : GenericSolver
{
    private Chromosome _initialValue = null;
    private Chromosome[] _population;
```

```
internal class EvolutionaryHillClimbing : GenericSolver
{
    private Chromosome _initialValue;
    private Chromosome[] _population;
```

d) Eliminarea variabilelor neutilizate

```
NeuralNet nn = new NeuralNet(nds, noNeurons);
nn.AutoConnect();
nn.InitWeightsAndBiases();
double err = nn.BackPropTrain(Settings.BPEpochs, Settings.BPErro
nn.FindOptimalTopology(Settings.DMSteps);
nn.SaveLog();
```

```
NeuralNet nn = new NeuralNet(nds, noNeurons);
nn.AutoConnect();
nn.InitWeightsAndBiases();
nn.BackPropTrain(Settings.BPEpochs, Settings.BPError, Settings.B
nn.FindOptimalTopology(Settings.DMSteps);
nn.SaveLog();
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

e) Optimizarea iniţializărilor în constructor

```
internal class NeuralNetwork
{
    public int[] activationFunctions;
    public double[] biases;
    public int inputs;
    public int maxH1;
    public int maxH2;
    public int noH1;
    public int noH2;
    public int noHiddenLayers;
    public int outputs;
    public double[] steepness;
    public Hashtable weights;

    public NeuralNetwork(Chromosome chr)
    {
        int i;
        int j;
        double w;
        string htkey;
        this.inputs = Settings.NoInputs;
        this.outputs = Settings.NoOutputs;
        this.maxH1 = Settings.MaxNeuronsH1;
        this.maxH2 = Settings.MaxNeuronsH2;
        this.noHiddenLayers = (int) chr.RealGenes[0];
```

```
internal class NeuralNetwork
{
    public int[] activationFunctions;
    public double[] biases;
    public int inputs = Settings.NoInputs;
    public int maxH1 = Settings.MaxNeuronsH1;
    public int maxH2 = Settings.MaxNeuronsH2;
    public int noH1;
    public int noH2;
    public int noHiddenLayers;
    public int outputs = Settings.NoOutputs;
    public double[] steepness;
    public Hashtable weights;

    public NeuralNetwork(Chromosome chr)
    {









        this.noHiddenLayers = (int) chr.RealGenes[0];
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

f) Optimizarea blocurilor switch

```
private double GetBiasedActivation(double netInput, int ne
{
    double outputResult = 0;
    netInput += this.biases[neuronIndex];
    switch (this.activationFunctions[neuronIndex])
    {
        case 1:
            if (netInput < 0)
                outputResult = -0.9;
            else
                outputResult = 0.9;
            break;

        case 2:
            if (netInput < -0.9)
                outputResult = -0.9;
            else if (netInput > 0.9)
                outputResult = 0.9;
            else
                outputResult = netInput;
            break;
```

```
private double GetBiasedActivation(double netInput, int ne
{

    netInput += this.biases[neuronIndex];
    switch (this.activationFunctions[neuronIndex])
    {
        case 1:
            if (netInput >= 0.0)
            {
                return 0.9;
            }
            return -0.9;

        case 2:
            if (netInput >= -0.9)
            {
                if (netInput > 0.9)
                {
                    return 0.9;
                }
                return netInput;
            }
            return -0.9;
```

Prin urmare, programatorul nu trebuie să facă optimizări, mai ales când acestea scad claritatea codului. Singurele optimizări recomandate sunt acelea care scad complexitatea unui algoritm cu cel puțin o clasă, de

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

exemplu de la $O(n^2)$ la $O(log\ n)$ sau $O(n)$. În rest, compilatorul face oricum transformări ale codului, adaptate mediului de execuție existent. Eventualele optimizări manuale pot conduce în cel mai rău caz la secvențe nestandard care nu sunt recunoscute de compilator şi care pot *scădea* de fapt performanțele aplicației.

Codul pregătit pentru livrarea comercială trebuie întotdeauna compilat în modul *Release*.

3.4. Obscurizarea codului. *Dotfuscator*

Codul obscurizat (engl. "obfuscated") este un cod foarte greu de citit şi de înțeles. Deoarece prin decompilare orice program .NET devine de fapt *open-source*, obscurizarea este una din modalitățile prin care se poate păstra secretul asupra codului aplicațiilor realizate.

Visual Studio include un astfel de instrument, numit *Dotfuscator Community Edition* care are o serie de limitări față de versiunea *Professional*. Printre cele mai importante sunt criptarea şirurilor de caractere, comprimarea *assembly*-urilor obscurizate şi diferite scheme de redenumire. Nu este un instrument infailibil, însă este util pentru aplicațiile de importanță medie.

Dotfuscator Community Edition poate fi pornit din mediul Visual Studio din meniul: *Tools → Dotfuscator Community Edition.*

Mai întâi se încarcă *assembly*-ul dorit, iar din tab-ul *Rename* se pot selecta *namespace*-urile, tipurile şi metodele care se doresc redenumite, implicit toate. Apoi se rulează proiectul apăsând butonul *Build* (figura 1.4).

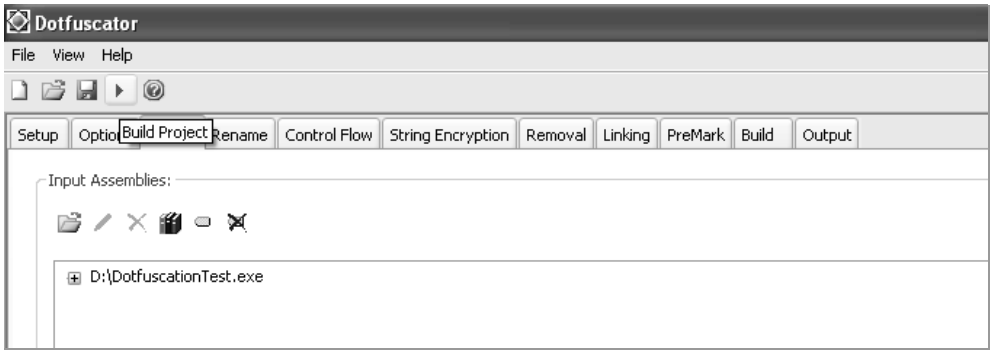


Figura 1.4. *Instrumentul Dotfuscator*

Rezultatul va fi un nou *assembly*, cu numele interne schimbate.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Să considerăm următorul exemplu. În stânga este programul inițial iar în dreapta codul dezasamblat după obscurizare.

```
public class AddingNumbers
{
   public int AddTwo(int a, int b)
   {
      return a + b;
   }

   public int AddThree(int a, int b, int c)
   {
      return a + b + c;
   }
}

class Program
{
   static void Main(string[] args)
   {
      int x = 1, y = 2, z = 3;
      AddingNumbers an =
            new AddingNumbers();
      int r1 = an.AddTwo(x, y);
      Console.WriteLine(r1);
      int r2 = an.AddThree(x, y, z);
      Console.WriteLine(r2);
   }
}
```

```
public class a
{
   public int a(int A_0, int A_1)
   {
      return (A_0 + A_1);
   }

   public int a(int A_0, int A_1, int A_2)
   {
      return ((A_0 + A_1) + A_2);
   }
}

class b
{
   private static void a(string[] A_0)
   {
      int num = 1;
      int num2 = 2;
      int num3 = 3;
      a a = new a();
      Console.WriteLine(a.a(num, num2));
      Console.WriteLine(
            a.a(num, num2, num3));
   }
}
```

4. Aplicații

4.1. Realizați un program de tip consolă în care să creați câte o metodă pentru fiecare din situațiile de mai jos. Compilați programul în mod *Debug* cu *Debug Info → full*, respectiv *Release* cu *Debug Info → none*.

- Variabile locale nefolosite:

```
int a = 4;
int b = 3;
double c = 4;
bool ok = false;
Console.WriteLine(ok);
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

- Ramuri ale expresiilor condiționale:

```
int b = 3;
double c = 4;
bool ok = false;

if (b < 3)
   if (c > 3)
      ok = true;

Console.WriteLine(ok);
```

- Cod imposibil de atins:

```
if (true)
   Console.WriteLine("ok");
if (false)
   Console.WriteLine("false");
else
   Console.WriteLine("true");
return;
Console.WriteLine("finished");
```

- Expresii aritmetice:

```
int a = 2 + 4 + 5;
double b = 9 / 5.0 + a + 9 + 5;
b++;
```

- Instrucțiuni goto (nerecomandate, deoarece afectează calitatea structurii codului):

```
int b = 10;
if (b < 20)
{
   Console.WriteLine("true");
}
else
{
   goto After;
}
After:
   Console.WriteLine("goto");
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    bool a = (b < 4);

    if (a)
    {
        goto C;
    }
    Console.WriteLine(1);
C:
    Console.WriteLine(2);
```

- Apelarea metodelor din obiecte (pentru aceasta creați o clasă Test cu o metodă MyMethod):

```
Test t = new Test();
int a = t.MyMethod();
Console.WriteLine(a);
```

- De observat decompilarea următoarei secvențe în mod *Release*: ce elemente își păstrează numele chiar în absența fișierului *pdb*? Care sunt numele implicite date de *.NET Reflector* pentru diferite tipuri de date?

```
int integer = 3;
double real = 3.14;
bool boolean = true;

Console.WriteLine("Integer: " + integer);
Console.WriteLine("Real: " + real);
Console.WriteLine("Boolean: " + boolean);

NumberForTestingOnly t = new NumberForTestingOnly();
Console.WriteLine("Test object: " + t.ReturnDouble(4));

public class NumberForTestingOnly
{
    public int ReturnDouble(int par)
    {
        return par * 2;
    }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Notă: deşi de multe ori termenii „argument” şi „parametru” se folosesc ca sinonime, există o diferență între aceste noțiuni. În exemplul anterior:

- ReturnDouble(**int** par) – par este parametru;
- t.ReturnDouble(4) – 4 este argument.

4.2. Creați un program cu 3 clase, fiecare clasă cu 4 metode şi obscurizați-l utilizând instrumentul *Dotfuscator*. Pentru a vedea rezultate interesante, câteva metode din aceeaşi clasă trebuie să aibă aceeaşi semnătură cu excepția numelui, iar celelalte să aibă semnături diferite. Instanțiați obiecte de aceste tipuri şi apelați metodele corespunzătoare.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 2

Stilul de scriere a codului. Tratarea excepţiilor. Interfaţa grafică cu utilizatorul

1. Obiective
2. Stilul de scriere a codului
3. Tratarea excepţiilor
4. Interfaţa cu utilizatorul
5. Interfaţa grafică cu utilizatorul în Microsoft Visual Studio .NET
6. Elemente de C#
7. Aplicaţii

1. Obiective

Obiectivele capitolului 2 sunt următoarele:

- Sublinierea importanței unui stil unitar de scriere a codului într-o firmă care dezvoltă produse software;
- Descrierea standardului de scriere a codului pe care îl vom utiliza în ghidul de aplicații de ingineria programării;
- Explicarea modalităților de tratare a excepțiilor în C#;
- Prezentarea unor aspecte legate de dezvoltarea de aplicații cu interfață grafică;
- Prezentarea modului de lucru cu proprietăți C#.

2. Stilul de scriere a codului

Unul din scopurile urmărite la scrierea programelor trebuie să fie întreținerea ulterioară a codului, adică facilitarea modificărilor şi completărilor viitoare, foarte probabil de către persoane diferite decât autorul inițial. De asemenea, unele studii au arătat că după 6 luni de la scrierea unui program, acesta îi apare la fel de străin autorului ca şi un program scris de altcineva.

Unul din aspectele principale ale codului uşor de întreținut este posibilitatea de a găsi anumite bucăți de cod şi de a le modifica fără a afecta celelalte secțiuni. Claritatea este esențială. Altfel, în cazul programelor de

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

mari dimensiuni, așa cum sunt majoritatea situațiilor în cazul produselor software comerciale, în locul lucrului efectiv pentru adăugarea de funcționalități se va pierde timpul încercându-se să se găsească porțiunile relevante de cod care trebuie modificate.

Formatarea codului poate simplifica înțelegerea structurii semantice sau poate cauza confuzie. Poate chiar ascunde defecte greu de depistat, de exemplu:

```
bool error = DoSomething();
if (error)
   Console.WriteLine("Eroare");
   Environment.Exit(1);              /// !!!
```

Nu contează cât de bine este proiectat un program; dacă prezentarea sa arată neglijent, va fi neplăcut de lucrat cu el.

2.1. Acoladele

Există două modalități principale de plasare a acoladelor.

Stilul Kernighan şi Ritchie este bazat pe dorința de a afișa cât mai multe informații într-un mod compact:

```
int KernighanRitchie() {
   int a = 0, b = 0;
   while (a != 10) {
      a++;
      b--;
   }
   return b;
}
```

Acest stil poate fi folosit la prezentări de cod sau în situații în care spațiul disponibil pentru afișarea codului este redus, de exemplu într-un material tipărit.

Stilul extins sau *stilul Allman* este recomandat de Microsoft pentru limbajul C#:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
int Extended()
{
    int a = 0, b = 0;
    while (a != 10)
    {
        a++;
        b--;
    }
    return b;
}
```

Avantajul principal al acestuia este claritatea, deoarece blocurile de cod sunt evidenţiate prin alinierea acoladelor. Este stilul pe care îl vom utiliza în acest ghid de aplicaţii.

2.2. Standarde de programare

Mulţi programatori fără experienţă industrială, deşi foarte buni, refuză la început aplicarea unor standarde impuse. Dacă programul este corect, ei nu înţeleg de ce trebuie aliniat altfel sau de ce trebuie schimbate numele variabilelor sau metodelor.

Este important de avut în vedere faptul că *nu există un stil perfect*, deci războaiele privind cea mai bună formatare nu pot fi câştigate. Toate stilurile au argumente pro şi contra. Majoritatea firmelor serioase de software au standarde interne de scriere a programelor, care definesc regulile pentru prezentarea codului. Aceste standarde cresc calitatea programelor şi sunt importante deoarece toate proiectele livrate în afara organizaţiei vor avea un aspect îngrijit şi coerent, de parcă ar fi fost scrise de aceeaşi persoană. Existenţa mai multor stiluri distincte într-un proiect indică lipsa de profesionalism.

Faptul că un programator crede că stilul său propriu este cel mai frumos şi cel mai uşor de înţeles nu are nicio importanţă. Un stil care unui programator îi pare în mod evident cel mai bun poate reprezenta o problemă pentru altul. De exemplu:

```
using System . Windows . Forms;
namespace LabIP {
  public class HelloWorld:System . Windows . Forms . Form {
    public HelloWorld ( ) {
    InitializeComponent ( );
      }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    protected override void Dispose ( bool disposing ) {
    if( disposing ) {
    if( components != null ) {
    components . Dispose ( );
      }
      }
    base . Dispose ( disposing );
      }
    static void Main ( ) {
    Application . Run ( new HelloWorld ( ) );
      }
    private void button1_Click ( object sender , System . EventArgs e ) {
    string STRING = "Hello World!";
    display ( STRING );
      }
    private void display ( string STR ) {
    MessageBox . Show ( STR , ":-)" );
      }
    }
  }
```

Acest program nu a fost aliniat aleatoriu, ci folosind opțiunile din mediul Visual Studio: *Tools → Options → Text Editor → C# → Formatting*. În acelaşi mod, stilul personal al unui programator poate părea la fel de ciudat altuia.

Beneficiile adoptării unui stil unitar de către toți membrii unei organizații depăşesc dificultățile inițiale ale adaptării la un stil nou. Chiar dacă nu este de acord cu standardul impus, un programator profesionist trebuie să se conformeze. După ce va folosi un timp stilul firmei, se va obişnui cu el şi i se va părea perfect natural.

În prezentul ghid de aplicații vom utiliza un standard bazat pe recomandările Microsoft pentru scrierea programelor C#.

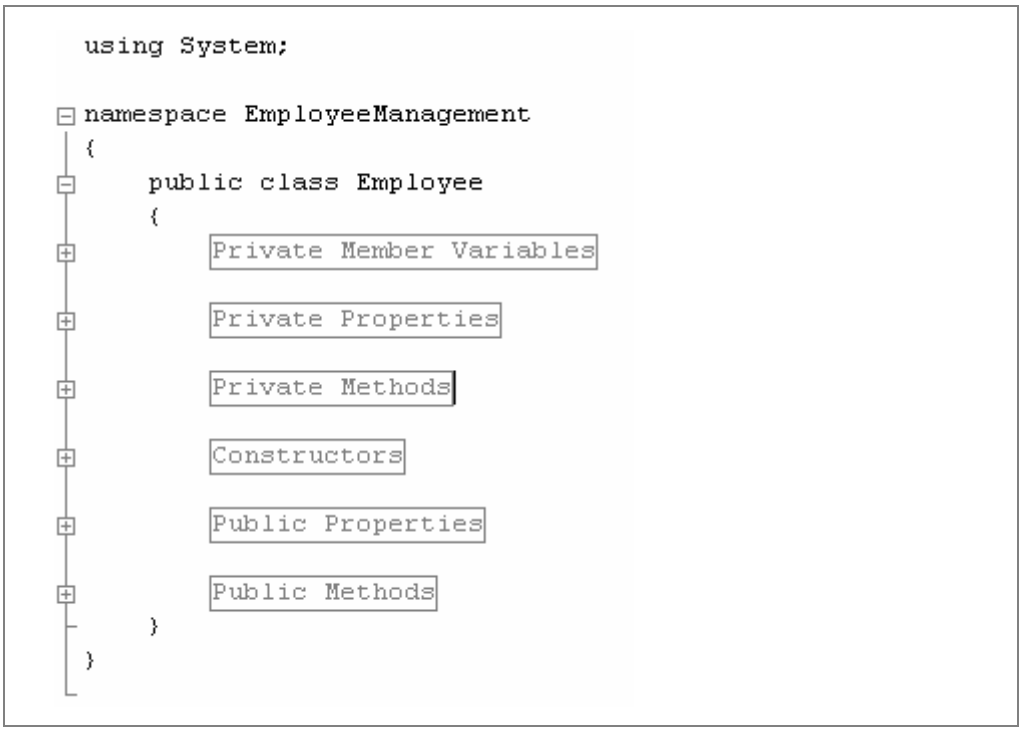
Este bine ca regiunile de program să fie delimitate cu ajutorul cuvântului cheie region, de exemplu:

```
#region Fields
private DateOfBirth _dob;
private Address _address;
#endregion
```

Fields

Dacă toate secțiunile unei clase sunt delimitate pe regiuni, pagina ar trebui să arate în felul următor atunci când toate definițiile sunt colapsate:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com



2.3. Convenţii pentru nume

Cheia alegerii unor nume potrivite este înţelegerea rolului construcţiilor respective. De exemplu, dacă nu putem găsi un nume bun pentru o clasă sau pentru o metodă, ar fi util să ne întrebăm dacă ştim într-adevăr la ce foloseşte aceasta sau dacă chiar este necesar să existe în program. Dificultăţile la alegerea unui nume potrivit pot indica probleme de proiectare.

Un nume trebuie să fie:

- *Descriptiv:* Oamenii îşi păstrează deseori percepţiile iniţiale asupra unui concept. Este importantă deci crearea unei impresii iniţiale corecte despre datele sau funcţionalităţile unui program prin alegerea unor termeni care să descrie exact semnificaţia şi rolul acestora. Numele trebuie alese din perspectiva unui cititor fără cunoştinţe anterioare, nu din perspectiva programatorului;
- *Adecvat:* Pentru a da nume clare, trebuie să folosim cuvinte din limbajul natural. Programatorii au tendinţa să utilizeze abrevieri şi prescurtări, însă acest lucru conduce la denumiri confuze. Nu are

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

importanță faptul că un identificator este lung câtă vreme este lipsit de ambiguitate. Denumirea st nu este o alegere potrivită pentru conceptul numarStudenti. Regula de urmat este: *trebuie preferată claritatea față de laconism*. Excepțiile sunt contoarele de bucle, de exemplu clasicul **for** (**int** i = 0; i < length; i++). Acestea de multe ori nu au o semnificație de sine stătătoare, sunt construcții specifice (idiomuri) ale limbajelor evoluate din C şi de obicei se notează cu o singură literă: i, j, k etc.;

- *Coerent:* Regulile de numire trebuie respectate în tot proiectul şi trebuie să se conformeze standardelor firmei. O clasă precum cea de mai jos nu prezintă nicio garanție de calitate:

```
class badly_named : MyBaseClass
{
    public void doTheFirstThing();
    public void DoThe2ndThing();
    public void do_the_third_thing();
}
```

Pentru descrierea tipurilor de nume, există în engleză o serie de termeni care nu au un echivalent exact în limba română:

- *Pascal case:* Primul caracter al tuturor cuvintelor este o majusculă iar celelalte caractere sunt minuscule, de exemplu: **N**umar**S**tudenti;
- *Camel case:* Pascal case cu excepția primului cuvânt, care începe cu literă mică, de exemplu: **n**umar**S**tudenti.

Pentru denumirea conceptelor dintr-un program C#, vom adopta convențiile din tabelul 2.1.

Tabelul 2.1. *Convenţii pentru denumirea conceptelor dintr-un program C#*

Concept	**Convenţie**	**Exemple**
Namespace-uri	Pascal case	namespace **LaboratorIP**
Clase	Pascal case	class **HelloWorld**
Interfețe	Pascal case precedat de *I*	interface **IEntity**
Metode	Pascal case	void **SayHello**()
Variabile locale	Camel case	int **totalCount** = 0;
Variabile booleene	Prefixate cu *is*	bool **isModified**;

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Concept	**Convenţie**	**Exemple**
Parametrii metodelor	Camel case	void SayHello(string **name**)
Câmpuri private [1]	Camel case precedat de *underscore*	string **_address**;
Proprietăţi [2]	Pascal case	**Address**
Constante, câmpuri *readonly* publice [3]	Pascal case	const int **MaxSpeed** = 100;
Controale pentru interfaţa grafică [4]	Camel case precedat de *tipul controlului*	**buttonOK** **checkBoxTrigonometric** **comboBoxFunction**
Excepţii	Pascal case cu terminaţia *Exception*	**MyException** **PolynomialException**

[1] Până la versiunea Visual Studio 6.0, programatorii utilizau în C++ notaţia maghiară a lui Simonyi pentru variabile, care indica şi tipul acestora, de exemplu nAge pentru int. Pentru variabilele membru se folosea prefixul m_, de exemplu m_nAge. Pentru limbajul C#, Microsoft nu mai recomandă utilizarea acestor notaţii. Pentru câmpurile private există câteva avantaje la prefixarea cu *underscore*:

- câmpurile clasei vor avea o notaţie diferită de variabilele locale;
- câmpurile clasei vor avea o notaţie diferită de parametrii metodelor, astfel încât se vor evita situaţiile de iniţializare de genul this.x = x, unde this.x este câmpul iar x este parametrul metodei;
- în *IntelliSense*, la apăsarea tastei _ vor apărea grupate toate câmpurile.

Avantajul utilizării acestei convenţii se manifestă mai ales în situaţii precum aceea de mai jos:

```
private int description; // ortografie corectă

public Constructor(int descripton) // ortografie incorectă pentru "description"
{
    this.description = description; // ortografie corectă în ambele părţi, câmpul rămâne 0
}
```

[2] Proprietăţile vor fi detaliate în secţiunea 6.2.

[3] În mare, tot ce e public într-o clasă trebuie să înceapă cu literă mare

[4] Această notaţie are avantajul că, deşi numele sunt mai lungi, sunt lipsite de ambiguitate. Există şi stilul prefixării cu o abreviere de 3 litere, de exemplu btn pentru Button, ckb pentru CheckBox etc. Pentru controale mai puţin uzuale, semnificaţiile prefixelor nu mai sunt evidente: pbx, rdo, rbl etc.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

3. Tratarea excepţiilor

Tratarea erorilor se făcea în C prin returnarea unei valori, de obicei int, care semnifica un cod de eroare. De exemplu, dacă o funcție trebuia să deschidă un fişier, ea putea întoarce 0 dacă totul a funcționat normal, respectiv codul de eroare 1 dacă fişierul nu exista. În funcția apelantă, programatorul trebuia să trateze codul returnat:

```
int err = OpenFile(s);
```

Cu toate acestea, programatorul era liber să apeleze funcția direct, OpenFile(s), fără a mai testa valoarea returnată. Programul putea fi testat cu succes, deoarece fişierul exista, însă putea să nu mai funcționeze după livrare.

Majoritatea funcțiilor *Windows API* returnează un cod dintr-o listă cu sute de valori posibile, însă puțini programatori testează aceste valori individual.

Tratarea excepțiilor a apărut pentru a da posibilitatea programelor să surprindă şi să trateze erorile într-o manieră elegantă şi centralizată, permițând separarea codului de tratare a erorilor de codul principal al programului, ceea ce face codul mai lizibil. Astfel, este posibilă tratarea:

- tuturor tipurilor de excepții;
- tuturor excepțiilor de un anume tip;
- tuturor excepțiilor de tipuri înrudite.

Odată ce o excepție este generată, ea nu poate fi ignorată de sistem. Funcția care detectează eroarea poate să nu fie capabilă să o trateze şi atunci se spune că „aruncă” (throw) o excepție. Totuşi, nu putem fi siguri că există un caz de tratare pentru orice excepție. Dacă există o rutină potrivită, excepția este tratată, dacă nu, programul se termină. Rutinele de tratare a excepțiilor pot fi scrise în diverse feluri, de exemplu examinează excepția şi apoi închid programul sau re-aruncă excepția.

Structura blocurilor try-catch pentru tratarea excepțiilor este următoarea:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
FuncţieApelată
{
    ...
    if (condiţii)
        throw ...
            ...
}

FuncţieApelantă
{
    ...
    try
    {
        FuncţieApelată();
    }
    catch (Exception e) // catch fără parametru dacă nu interesează detaliile excepţiei
    {
        // cod pentru tratarea excepţiei, care prelucrează parametrul e
    }
    ...
}
```

Codul care ar putea genera o excepție se introduce în blocul try. Excepția este aruncată (throw) din funcția care a fost apelată, direct sau indirect. Excepțiile care apar în blocul try sunt captate în mod normal de blocul catch care urmează imediat după acesta. Un bloc try poate fi urmat de unul sau mai multe blocuri catch. Dacă se execută codul dintr-un bloc try şi nu se aruncă nicio excepție, toate rutinele de tratare sunt ignorate, iar execuția programului continuă cu prima instrucțiune de după blocul (sau blocurile) catch.

În C#, tipul trimis ca parametru este de obicei Exception:

```
try
{
    FunctieApelata();
}
catch (Exception ex)
{
    ...
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

De cele mai multe ori, ne interesează proprietatea Message (de tip string) a unui astfel de obiect, care arată cauza erorii. Textul respectiv este precizat în funcția care aruncă excepția.

```
private void FunctieApelata(int a)
{
    if (a == 0)
        throw new Exception("Argument zero");
}

private void FunctieApelanta()
{
    FunctieApelata(0);
}
```

Dacă excepția nu este tratată, va apărea un mesaj de atenționare cu textul dorit (figura 2.1).

Figura 2.1. *Mesaj de excepție*

Mesajul de eroare trebuie preluat în program în blocul catch:

```
private void FunctieApelanta()
{
    try
    {
        FunctieApelata(0);
    }
    catch (Exception ex)
    {
        // ex.Message este "Argument zero"
    }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Programatorul îşi poate defini propriile tipuri de excepţii, în cazul în care are nevoie să trimită informaţii suplimentare privind excepţia. Acestea trebuie însă derivate din clasa Exception:

```
public class MyException: Exception
{
    public int _info; // informaţie suplimentară

    // în constructor se apelează şi constructorul clasei de bază
    public MyException(int val) : base()
    {
        _info = val;
    }
}
```

Utilizarea acestui tip se face astfel:

```
throw new MyException(3);
```

După un try pot exista mai multe blocuri catch. Ele trebuie dispuse în ordinea inversă a derivării tipurilor, de la particular la general:

```
try
{
    FunctieApelata(x);
}
catch (MyException myex)
{
    ...
}
catch (Exception ex)
{
    ...
}
```

Altă ordine nu este permisă, eroarea fiind descoperită la compilare: *A previous catch clause already catches all exceptions of this or a super type ('System.Exception')*.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

3.1. Tratarea excepţiilor pe firul de execuţie al aplicaţiei

Uneori pentru a fi siguri că nu am lăsat vreo excepție netratată, putem trata global toate excepțiile apărute pe firele de execuție ale aplicației, în maniera descrisă în continuare:

```
static class Program
{
   static void Main()
   {
      // Adăugarea unui event handler pentru prinderea excepţiilor
      // din firul principal al interfeţei cu utilizatorul
      Application.ThreadException +=
            new ThreadExceptionEventHandler(OnThreadException);

      // Adăugarea unui event handler pentru toate firele de execuţie din appdomain
      // cu excepţia firului principal al interfeţei cu utilizatorul
      AppDomain.CurrentDomain.UnhandledException +=
            new UnhandledExceptionEventHandler(CurrentDomain_UnhandledException);

      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault(false);
      Application.Run(new MyForm());   // MyForm este fereastra principală a programului
   }

   // Tratează excepţiile din firul principal al interfeţei cu utilizatorul
   static void OnThreadException(object sender, ThreadExceptionEventArgs t)
   {
      // Afişează detaliile excepţiei
      MessageBox.Show(t.Exception.ToString(), "OnThreadException");
   }

   // Tratează excepţiile din toate celelalte fire de execuţie
   static void CurrentDomain_UnhandledException(object sender,
      UnhandledExceptionEventArgs e)
   {
      // Afişează detaliile excepţiei
      MessageBox.Show(e.ExceptionObject.ToString(), "CurrentDomain_UnhandledException");
   }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

4. Interfaţa cu utilizatorul

Interfaţa cu utilizatorul permite acestuia să interacţioneze cu aplicaţia. Există două tipuri principale de interfeţe:

- interfaţă cu utilizatorul de tip *caracter* (engl. "character user interface", *CUI*);
- interfaţă cu utilizatorul de tip *grafic* (engl. "graphical user interface", *GUI*).

Un exemplu de interfaţă de tip caracter este interfaţa la linia de comandă a sistemului de operare MS-DOS. Când se foloseşte o astfel de interfaţă, în general utilizatorul trebuie să memoreze şi să tasteze comenzi text. De aceea, interfeţele de acest tip pot fi mai dificil de utilizat şi necesită o oarecare pregătire a utilizatorului. O interfaţă grafică încearcă să simplifice comunicarea. Graficele reprezintă obiecte pe care utilizatorul le poate manipula şi asupra cărora poate efectua acţiuni. Deoarece utilizatorul nu trebuie să ştie un limbaj de comandă, o interfaţă grafică bine proiectată este mai uşor de folosit decât o interfaţă de tip caracter.

4.1. Proiectarea comenzilor şi interacţiunilor

Dacă ierarhia de comenzi trebuie să se integreze într-un sistem de interacţiuni deja existent, trebuie mai întâi studiat acesta.

Se stabileşte o ierarhie iniţială de comenzi care poate fi prezentată utilizatorilor în mai multe moduri: o serie de opţiuni dintr-un meniu, o bară de instrumente (*toolbar*) sau o serie de imagini (*icons*).

Apoi, această ierarhie se rafinează prin ordonarea serviciilor din fiecare ramură a ierarhiei în ordinea logică în care trebuie să se execute, cele mai frecvente servicii apărând primele în listă.

Lăţimea şi adâncimea ierarhiei trebuie proiectate în aşa fel încât să se evite supraîncarea memoriei de scurtă durată a utilizatorului. De asemenea, trebuie minimizat numărul de paşi sau de acţiuni (apăsări ale *mouse*-ului, combinaţii de taste) pe care trebuie să le efectueze acesta pentru a-şi îndeplini scopul.

Interacţiunile cu factorul uman pot fi proiectate pe baza următoarelor criterii:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

- *coerența*: se recomandă utilizarea unor termeni şi acțiuni cu semnificații unice, bine precizate şi care se regăsesc în mod unitar în tot sistemul;
- *numărul mic de paşi*: trebuie să se minimizeze numărul de acțiuni pe care trebuie să le îndeplinească utilizatorul;
- *evitarea „aerului mort"* (engl. "dead air"): utilizatorul nu trebuie lăsat singur, fără niciun semnal, atunci când aşteaptă ca sistemul să execute o acțiune. El trebuie să ştie că sistemul execută o acțiune şi cât din acțiunea respectivă s-a realizat;
- *operațiile de anulare* (engl. "undo"): se recomandă furnizarea acestui serviciu datorită erorilor inerente ale utilizatorilor;
- *timpul scurt şi efortul redus de învățare*: de multe ori, utilizatorii nu citesc documentația, de aceea se recomandă furnizarea în timpul execuției a unor soluții pentru problemele apărute;
- *aspectul estetic al interfeței:* oamenii utilizează mai uşor un produs software cu un aspect plăcut.

Se recomandă folosirea de icoane şi controale similare celor din produsele software cu care utilizatorul este familiarizat. Dacă are de-a face cu acelaşi aspect exterior, acesta îşi va folosi cunoştințele anterioare pentru navigarea prin opțiunile programului, ceea ce va reduce şi mai mult timpul de instruire.

4.2. Considerente practice

Elementele interfeței grafice trebuie să fie coerente, adică să aibă stiluri, culori şi semnificații similare în toată aplicația. Un tabel centralizat de cuvinte cheie poate ajuta la alegerea textelor sau etichetelor de către proiectanții interfeței care lucrează la acelaşi sistem. Aceast tabel conține lista de cuvinte folosite în toată interfața şi care înseamnă aceeaşi lucru peste tot.

O interfață clară este uşor de înțeles. Metaforele utilizate trebuie să fie în acord cu experiența utilizatorilor în legătură cu obiectele din lumea reală pe care le reprezintă. De exemplu, icoana unui coş de gunoi poate reprezenta o funcție de gestionare a fişierelor nedorite (gen *Recycle Bin*): fişierele pot fi introduse în coş, unde rămân şi pot fi regăsite până când coşul este golit.

Erorile trebuie identificate imediat folosind un mesaj inofensiv. Utilizatorul trebuie să-şi poată repara o greşeală ori de câte ori este posibil acest lucru iar documentația programului sau manualul de utilizare trebuie

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

să îl ajute în acest sens. Mesajele critice trebuie folosite numai atunci când utilizatorul trebuie avertizat că unele date pot fi pierdute sau deteriorate dacă nu acționează imediat.

Chiar dacă nu sunt probleme, furnizarea informațiilor despre starea sistemului face interfața mai prietenoasă. Totuşi, aceste mesaje pot deveni supărătoare dacă sunt folosite prea des. Pot fi afişate şi mesaje despre cursul unei acțiuni sau despre timpul după care se va termina o activitate. Dacă o acțiune durează aproximativ 6-8 secunde, se poate folosi un indicator de tip clepsidră. Pentru activități mai lungi de 8 secunde, se pot folosi indicatoare de tip procentaj sau timp rămas. Utilizatorii trebuie să aibă de asemenea posibilitatea întreruperii sau anulării acestor acțiuni de durată.

Mesajele sonore trebuie în general evitate deoarece pot fi supărătoare, distrag atenția iar semnificația sunetelor poate depinde de contextul cultural al utilizatorilor.

Culoarea joacă un rol important în proiectarea unei interfețe grafice. Folosirea corectă a culorilor face interfața clară şi uşor de navigat. Totuşi, dacă nu sunt folosite cu atenție, culorile pot distrage atenția utilizatorului.

Culorile pot fi utilizate pentru a identifica părțile importante ale interfeței. Prea multe culori strălucitoare fac textul dificil de citit. Trebuie de asemenea evitat un fundal complet alb şi nu trebuie folosite mai mult de 4 culori într-o fereastră. Interpretarea culorilor este foarte subiectivă. Poate depinde de asociații culturale, psihologice şi individuale. Deci, în general, cel mai bine este să folosim culori subtile şi neexagerate. Utilizatorii presupun de multe ori că există o legătură între obiectele de aceeaşi culoare, aşa că trebuie să fim atenți să nu folosim aceeaşi culoare pentru obiecte fără legătură.

Culorile nu trebuie să fie singura sursă de informații deoarece unii utilizatori nu pot distinge anumite culori, iar alții pot avea monitoare care nu suportă o gamă largă de culori.

Ca şi culorile, **icoanele** pot pune în valoare o interfață grafică, dacă sunt folosite corect. Icoanele bine proiectate oferă utilizatorului un mod accesibil de comunicare cu aplicația. Există câteva elemente de care trebuie să ținem seama la proiectarea acestora:

- un stil şi o dimensiune comună pentru toate icoanele dau interfeței un aspect coerent;
- desenele ajută utilizatorul să recunoască metaforele şi să-şi amintească funcțiile;

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

- conturarea icoanelor cu negru le face să iasă în evidență din fundal;
- icoanele pot fi afişate în trei mărimi: 48 x 48, 32 x 32 şi 16 x 16 pixeli; acestea trebuie să fie uşor de recunoscut chiar şi atunci când sunt afişate la dimensiunea de 16 x 16 pixeli;
- deşi o icoană poate fi accentuată prin culoare, ea trebuie să poată fi recunoscută şi în varianta alb-negru.

Icoanele bine proiectate îşi comunică funcțiile cu claritate şi cresc utilizabilitatea interfeței, însă o icoană neclară poate deruta utilizatorii şi poate creşte numărul de erori. Putem folosi etichete text pentru a ne asigura că semnificația unei icoane este clară. Câteodată este cel mai bine să folosim imagini tradiționale deoarece utilizatorul este familiarizat cu ele. O icoană bine proiectată trebuie să poată fi distinsă cu uşurință de celelalte icoane din jurul său, iar imaginea trebuie să fie simplă şi potrivită contextului interfeței.

Corpurile de literă (*font*-urile) utilizate într-o interfață grafică nu trebuie amestecate. Ca şi în cazul culorilor, prea multe *font*-uri pot distrage atenția utilizatorului.

4.3. Profilurile utilizatorilor

Pentru a crea o interfață grafică utilizabilă, trebuie să cunoaştem profilul utilizatorului, care descrie aşteptările şi nevoile acestuia. Un mod potrivit de a determina profilul utilizatorului este prin observare la locul său de muncă. Poate fi folositoare sugestia ca utilizatorul „să gândească cu voce tare" atunci când lucrează cu prototipul unei interfețe.

Aproape întotdeauna va exista un procent de utilizatori începători iar interfața trebuie să aibă grijă de aceştia. De exemplu, putem asigura alternative la acceleratori (combinații de taste pentru anumite funcții) şi putem preciza *shortcut*-urile în opțiunile meniurilor.

Profilurile utilizatorilor se încadrează în general în trei categorii:

- *Utilizatorul comod* doreşte să folosească interfața imediat, cu foarte puțin antrenament. Acest tip de utilizator preferă utilizarea *mouse*-ului, atingerea sensibilă a ecranului sau stiloul electronic. Navigarea simplă este importantă deoarece utilizatorul comod nu ține minte căi complicate. Afişarea unei singure ferestre la un moment dat simplifică navigarea. Pentru a face o interfață grafică accesibilă unui utilizator de acest tip, ea trebuie să se bazeze pe

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

recunoaşterea unei icoane, mai degrabă decât pe amintirea a ceea ce reprezintă icoana. Acest lucru se poate realiza prin folosirea unei multitudini de grafice şi de opţiuni în meniuri;

- *Utilizatorul rapid* doreşte un timp de răspuns cât mai mic, aşa încât trebuie evitate prea multe redesenări ale ferestrelor. Acest tip de utilizator preferă în general să folosească tastatura şi mai puţin *mouse*-ul. Utilizatorii de acest tip au în general timp pentru instruire şi sunt dispuşi să renunţe la facilităţi în favoarea vitezei. Acceleratorii le permit să lucreze mai repede;
- *Utilizatorul energic* este de nivel avansat şi are experienţă cu interfeţele grafice. Acesta nu îşi doreşte o instruire de durată şi se aşteaptă să folosească interfaţa imediat. Deoarece este sigur pe sine şi îi place să exploreze, trebuie întotdeauna asigurată o opţiune de anulare (engl. "undo"). Alte trăsături pe care le aşteaptă sunt schimbări limitate ale modului de afişare, multitasking şi posibilitatea particularizării şi individualizării aspectului interfeţei grafice.

5. Realizarea programelor cu interfaţa grafică cu utilizatorul în Microsoft Visual Studio .NET

Când este creat un nou proiect C# de tip *Windows Application*, în mijlocul ecranului (figura 2.2), apare un „formular" (Form) – fereastra principală a programului, în care se vor adăuga diverse componente de control: butoane, *text-box*-uri etc.

În partea din stânga a ecranului există o bară de instrumente (*View → Toolbox* sau *Ctrl+Alt+X*) din care se aleg cu *mouse*-ul componentele ce trebuie adăugate în fereastră.

Pentru adăugarea unei componente, programatorul va face click cu *mouse*-ul pe imaginea corespunzătoare din *toolbox*, apoi va face click în formular, în locul unde doreşte să apară componenta respectivă. Odată introduse în fereastră, componentele pot fi mutate, redimensionate, copiate sau şterse. În dreapta este o fereastră de proprietăţi (*View → Properties Window* sau *F4*). De aici, fiecărei componente folosite i se pot modifica proprietăţile, adică aspectul exterior, aşa cum va apărea în program, sau caracteristicile funcţionale interne. De asemenea, se pot selecta evenimentele corespunzătoare componentei care vor fi tratate în program.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

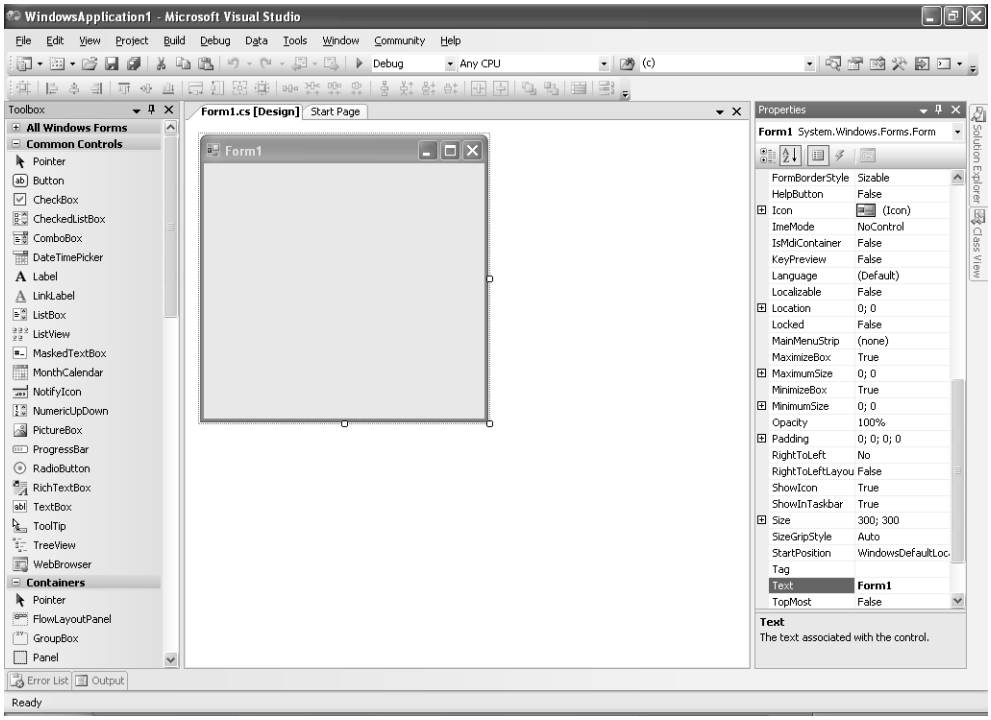


Figura 2.2. *Mediul de dezvoltare Microsoft Visual Studio*

În continuare, vor fi prezentate câteva componente de control folosite practic în orice program Windows. Pentru fiecare componentă, vor fi amintite unele proprietăţi şi metode uzuale. Pentru o descriere mai amănunţită, se recomandă consultarea documentaţiei *MSDN*.

Application

Clasa Application încapsulează o aplicaţie Windows. Clasa conţine metode şi proprietăţi statice pentru managementul unei aplicaţii, cum ar fi metode pentru pornirea şi oprirea programului, prelucrarea mesajelor Windows şi proprietăţi corespunzătoare informaţiilor despre aplicaţie.

Se poate observa că în scheletul de program creat implicit de mediul de dezvoltare, în metoda Main() este pornit programul pe baza clasei corespunzătoare ferestrei principale:

```
Application.Run(new MainForm());
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Form

Clasa System.Windows.Forms.Form corespunde unei ferestre standard. O aplicaţie poate avea mai multe ferestre – una principală, câteva secundare şi câteva ferestre de dialog.

Unele proprietăţi:

- Icon – icoana care apare în bara de titlu a ferestrei;
- FormBorderStyle – înfăţişarea şi comportamentul chenarului, de exemplu, dacă fereastra poate fi redimensionată;
- Text – titlul ferestrei, care apare în bara de titlu şi în *taskbar*;
- StartPosition – locul unde apare fereastra pe ecran;
- Size – dimensiunea (înălţimea şi lăţimea ferestrei); de obicei se stabileşte prin redimensionarea ferestrei cu *mouse*-ul, în procesul de proiectare.

Câteva evenimente:

- Load, Closed – pentru diverse iniţializări în momentul creării ferestrei sau prelucrări în momentul închiderii acesteia;

În general, pentru tratarea unui eveniment în C#, este selectat mai întâi obiectul de tipul dorit (la noi fereastra), apoi în fereastra de proprietăţi se alege *tab*-ul de evenimente şi se identifică evenimentul căutat.

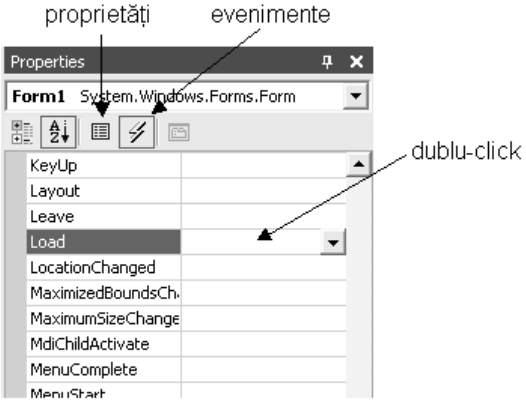


Figura 2.3. *Editarea proprietăţilor componentelor*

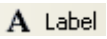
Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

După un dublu-click, ca în figura 2.3, se va crea automat o nouă metodă vidă corespunzătoare evenimentului, iar utilizatorul va trebui numai să scrie în corpul funcției comenzile dorite.

Button

Clasa System.Windows.Forms.Button corespunde unui buton. Câteva proprietăți şi evenimente:

- Text – textul înscris pe buton;
- Click – funcția executată când butonul este apăsat.

Label 

Clasa System.Windows.Forms.Label înscrie un text undeva în fereastră. Una din proprietăți:

- Text – textul înscris.

TextBox

Clasa System.Windows.Forms.TextBox corespunde unei căsuțe de editare de text. Câteva proprietăți şi evenimente:

- Text – textul din căsuță (de tip string);
- Multiline – textul poate fi introdus pe o singură linie (*false*) sau pe mai multe (*true*);
- ScrollBars – indică prezența unor bare de derulare (orizontale, verticale) dacă proprietatea Multiline este *true*;
- Enabled – componenta este activată sau nu (*true / false*);
- ReadOnly – textul poate fi modificat sau nu de utilizator (*true / false*);
- CharacterCasing – textul poate apărea normal (*Normal*), numai cu litere mici (*Lower*) sau numai cu litere mari (*Upper*);
- TextChanged – evenimentul de tratare a textului în timp real, pe măsură ce acesta este introdus.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

ComboBox

Clasa System.Windows.Forms.ComboBox corespunde unui *combo-box*, care combină un *text-box* cu o listă. Câteva proprietăţi şi evenimente:

- Text – textul din partea de editare;
- Items – lista de obiecte din partea de selecţie, care se poate introduce şi prin intermediul ferestrei de proprietăţi;
- SelectedIndex – numărul articolului din listă care este selectat (0 – primul, 1 – al doilea, etc., –1 dacă textul din partea de editare nu este ales din listă);
- TextChanged, SelectedIndexChanged – evenimente de tratare a schimbării textului prin introducerea directă a unui nou cuvânt sau prin alegerea unui obiect din listă.

MenuStrip

Clasa System.Windows.Forms.MenuStrip corespunde meniului principal al unei ferestre. Mod de folosire:

- se introduce o componentă de acest tip în fereastră;
- se editează meniul, direct în fereastră sau folosind proprietăţile;
- pentru separatori se introduce în câmpul Caption un minus („ – ”);
- literele care se vor a fi subliniate trebuie precedate de „&”;
- pentru implementarea metodei de tratare a unei opţiuni din meniu se va face dublu-click pe aceasta (sau pe evenimentul Click în fereastra de proprietăţi).

Timer

Clasa System.Windows.Forms.Timer încapsulează funcţiile de temporizare din Windows. Câteva proprietăţi şi evenimente:

- Tick – evenimentul care va fi tratat o dată la un interval de timp;
- Interval – intervalul de timp (în milisecunde) la care va fi executat codul corespunzător evenimentului Tick;
- Enabled – indică dacă *timer*-ul e activat sau nu (*true / false*).

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

string

Este o clasă care permite lucrul cu şiruri de caractere.
Există operatorul +, care permite concatenarea şirurilor:

```
string str1 = "Microsoft ";
string str2 = "Word";
str1 = str1 + str2; // sau str1 += str2; => str1 == "Microsoft Word"
```

Clasa conține multe proprietăți şi metode utile, dintre care amintim:

- **int** Length – lungimea şirului;
- **int** IndexOf(...) – poziția în şir la care apare prima dată un caracter sau un subşir;
- **string** Substring(...) – returnează un subşir;
- **string** Remove(**int** startIndex, **int** count) – returnează şirul rezultat prin ştergerea a *count* caractere din şir, începând cu poziția *startIndex*;
- **string**[] Split(...) – împarte şirul în mai multe subşiruri delimitate de anumite secvențe de caractere.

O metodă statică a clasei este Format(...), care returnează un şir de caractere corespunzător unui anumit format. Sintaxa este asemănătoare cu cea a funcției *printf* din C. De exemplu:

```
double d = 0.5;
string str = string.Format("Patratul numarului {0} este {1}", d, d * d);
```

Acelaşi rezultat s-ar fi putut obține astfel:

```
str = "Patratul numarului " + d.ToString() + " este " + (d * d).ToString();
```

Orice obiect are metoda ToString(), care convertește valoarea sa într-un şir. Pentru obiecte definite de programator, această metodă poate fi suprascrisă.

Dacă în exemplul de mai sus $d = 0.72654$ şi dorim să afişăm numerele numai cu 2 zecimale, metoda Format îşi dovedeşte utilitatea (figura 2.4).

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
string str=string.Format("Patratul numarului {0:F2} este {1:F2}", d, d*d);
MessageBox.Show(str);
```

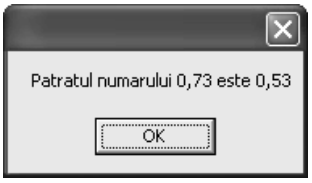


Figura 2.4. *Mesaj cu numere formatate*

StringBuilder

Majoritatea programatorilor utilizează de obicei clasa string când operează cu şiruri de caractere. În cazul concatenării unor şiruri, folosirea clasei StringBuilder aduce o importantă creştere de performanță. Să considerăm următorul bloc:

```
string listaNumere = "";
for (int i=0; i<1000; i++)
   listaNumere = listaNumere + " " + (i + 1).ToString();
```

În acest caz, se creează un nou obiect string la fiecare atribuire, adică de 1000 de ori!

Codul echivalent folosind StringBuilder este următorul:

```
StringBuilder sbListaNumere = new StringBuilder(10000);
for (int i = 0; i < 1000; i++)
{
   sbListaNumere.Append(" ");
   sbListaNumere.Append((i+1).ToString());
}
string listaNumere2 = sbListaNumere.ToString();
```

Pentru a avea acces la clasa StringBuilder, trebuie inclus *namespace*-ul System.Text la începutul programului:

```
using System.Text;
```

Directivele using sunt asemănătoare directivelor #include din C/C++. *Namespace*-urile sunt colecţii de clase predefinite sau definite de

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

programator. Dacă nu se dorește utilizarea unui *namespace*, clasa trebuie referită cu tot cu *namespace*. În cazul de față:

```
System.Text.StringBuilder sbListaNumere = new System.Text.StringBuilder(10000);
```

Ca dezavantaje, trebuie să menționăm că la inițializarea unui obiect StringBuilder performanțele scad. De asemenea, sunt foarte multe operații care nu se pot efectua cu StringBuilder, ci cu string. Pentru concatenarea a 2 şiruri nu este nevoie de StringBuilder. Se poate recomanda utilizarea acestei clase dacă e nevoie de concatenarea a cel puțin 4 şiruri de caractere. De asemenea, pentru creşterea vitezei, trebuie să estimăm lungimea finală a şirului (parametrul constructorului). Dacă folosim constructorul vid sau lungimea estimată este mai mică decât în realitate, alocarea spațiului se face automat iar performanța scade.

6. Elemente de C#

6.1. Clase parţiale

În clasele corespunzătoare ferestrelor, există metode predefinite generate automat de mediul Visual Studio, care conțin inițializarea controalelor grafice. Aceste metode sunt de obicei de dimensiuni mai mari şi sunt separate de logica aplicației fiind plasate într-un alt fişier, numit *.Designer.cs.

Definiția unei clase, structuri sau interfețe se poate împărți în două sau mai multe fişiere sursă. Fiecare fişier sursă conține o parte din definiția clasei şi toate părțile sunt combinate când aplicația este compilată. Există câteva situații când este utilă împărțirea definiției clasei:

- Când se lucrează la proiecte mari, împărțirea clasei în fişiere separate permite mai multor programatori să lucreze la ele simultan;
- Când se lucrează cu surse generate automat, codul poate fi adăugat clasei fără să mai fie nevoie de recrearea fişierului sursă. Visual Studio foloseşte această cale când creează ferestrele Windows, codul *wrapper* pentru servicii web ş.a.m.d. Programatorul poate scrie codul care foloseşte aceste porțiuni fără să mai fie nevoie de modificarea fişierului creat de Visual Studio.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Pentru a împărţi definiţia unei clase, se foloseşte cuvântul cheie partial, după cum este prezentat mai jos:

```
public partial class Student
{
   public void Learn()
   {
   }
}

public partial class Student
{
   public void TakeExam()
   {
   }
}
```

Modificatorul partial nu este valabil în declaraţiile pentru funcţii delegat sau enumerări.

6.2. Proprietăţi. Accesori

Proprietăţile sunt membri care furnizează un mecanism flexibil de citire, scriere, sau calculare de valori ale câmpurilor private. Ele pot fi folosite ca şi cum ar fi membri publici, dar ele sunt de fapt metode speciale, care permit ca datele să fie accesate mai uşor însă în acelaşi timp păstrează siguranţa şi flexibilitatea metodelor.

Proprietăţile combină aspecte specifice atât câmpurilor cât şi metodelor. Pentru utilizatorul unui obiect, o proprietate apare ca un câmp, deoarece accesul la proprietate necesită exact aceeaşi sintaxă. Pentru realizatorul clasei, o proprietate este formată din unul sau două blocuri de cod, reprezentând un accesor get şi/sau un accesor set. Blocul de cod pentru accesorul get este executat când proprietatea este citită, iar blocul de cod pentru accesorul set este executat când proprietăţii îi este atribuită o valoare.

O proprietate fără accesorul set este considerată *doar pentru citire* (*read-only)*. O proprietate fără accesorul get este considerată *doar pentru scriere* (*write-only)*. O proprietate cu ambii accesori este pentru *citire şi scriere (read-write)*.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

6.2.1. Accesorul *get*

Corpul accesorului get este similar cu cel al unei metode şi trebuie să returneze o valoare de tipul proprietății. Execuția accesorului get este echivalentă cu citirea valorii câmpului corespunzător. În următorul exemplu, este prezentat un accesor get ce întoarce valoarea unui câmp privat _name:

```
class Person
{
   private string _name;    // câmp

   public string Name       // proprietate
   {
      get
      {
         return _name;
      }
   }
}
```

Din exteriorul clasei, citirea valorii proprietății se realizează astfel:

```
Person p1 = new Person();
Console.Write(p1.Name);  // se apelează accesorul get
```

Accesorul get trebuie să se termine cu instrucțiunea return sau throw, iar fluxul de control nu poate depăşi corpul accesorului.

Când se întoarce doar valoarea unui câmp privat şi sunt permise optimizările, apelul către accesorul get este tratat *inline* de către compilator în aşa fel încât să nu se piardă timp prin efectuarea unui apel de metodă. Totuşi, un accesor get virtual nu poate fi tratat *inline* întrucât compilatorul nu ştie în timpul compilării ce metodă va fi de fapt apelată în momentul rulării.

Schimbarea stării unui obiect utilizând accesorul get nu este recomandată. De exemplu, următorul accesor produce ca efect secundar schimbarea stării obiectului de fiecare dată când este accesat câmpul _number.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
private int _number;

public int Number
{
   get
   {
      return _number++;      // !!! nerecomandat
   }
}
```

Accesorul get poate fi folosit fie ca să returneze valoarea unui câmp, fie să o calculeze şi apoi să o returneze. De exemplu:

```
class Student
{
   private string _name;

   public string Name
   {
      get
      {
         return _name != null ? _name : "NA";
      }
   }
}
```

În secvența de cod anterioară, dacă _name este null, proprietatea va întoarce totuşi o valoare, şi anume *"NA"* (engl. "No Account").

6.2.2. Accesorul *set*

Accesorul set (numit şi *mutator*) este similar cu o metodă ce întoarce void. Foloseşte un parametru implicit numit value (valoare), al cărui tip este tipul proprietății. În exemplul următor, este adăugat un accesor set proprietății Name:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
class Person
{
   private string _name;

   public string Name
   {
      get
      {
         return _name;
      }
      set
      {
         _name = value;
      }
   }
}
```

Când se atribuie o valoare proprietății, este apelat accesorul set cu un argument ce furnizează noua valoare. De exemplu:

```
Person p1 = new Person();
p1.Name = "Joe";  // se apelează accesorul set cu value = "Joe"
```

6.2.3. Aspecte mai complexe ale lucrului cu proprietăți

În exemplul următor, clasa TimePeriod reține o perioadă de timp în secunde, dar are o proprietate numită Hours care permite unui client să lucreze cu timpul în ore.

```
class TimePeriod
{
   private double _seconds;

   public double Hours
   {
      get { return _seconds / 3600; }
      set { _seconds = value * 3600; }
   }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
class Program
{
   static void Main()
   {
      TimePeriod t = new TimePeriod();

      // atribuirea proprietăţii Hours determină apelul accesorului set
      t.Hours = 24;

      // evaluarea proprietăţii Hours determină apelul accesorului get
      Console.WriteLine("Timpul in ore: " + t.Hours);
   }
}
```

În mod implicit, accesorii au aceeaşi vizibilitate, sau nivel de acces cu al proprietății căreia îi aparțin. Totuşi, uneori li se poate restricționa accesul. De obicei, aceasta implică restricționarea accesibilității accesorului set, în timp ce accesorul get rămâne public. De exemplu:

```
public string Name
{
   get
   {
      return _name;
   }
   protected set
   {
      _name = value;
   }
}
```

Aici, accesorul get primeşte nivelul de accesibilitate al proprietății însăşi, public în acest caz, în timp ce accesorul set este restricționat în mod explicit aplicând modificatorul de acces protected.

Proprietățile au mai multe utilizări: pot valida datele înainte de a permite o modificare, pot returna date când acestea sunt de fapt provenite din alte surse, precum o bază de date, pot produce o acțiune când datele sunt modificate, cum ar fi invocarea unui eveniment sau schimbarea valorii altor câmpuri.

În exemplul următor, Month este o proprietate:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
public class Date
{
   private int _month = 7;

   public int Month
   {
      get
      {
         return _month;
      }
      set
      {
         if ((value > 0) && (value < 13))
         {
            _month = value;
         }
      }
   }
}
```

Aici, accesorul set garantează că valoarea câmpului _month este între 1 şi 12. Locația reală a datelor proprietății este de obicei denumită „memorie auxiliară” (engl. “backing store”). Marea majoritate a proprietăților utilizează în mod normal câmpuri private ca memorie auxiliară. Câmpul este marcat privat pentru a se asigura faptul că nu poate fi accesat decât prin apelarea proprietății.

O proprietate poate fi declarată statică folosind cuvântul cheie static. În acest fel proprietatea devine disponibilă apelanților oricând, chiar dacă nu există nicio instanță a clasei.

Spre deosebire de câmpuri, proprietățile nu sunt clasificate ca variabile şi de aceea nu se poate transmite o proprietate ca parametru ref sau out.

Important! Există instrumente care generează automat proprietăți *read-write* pentru toate câmpurile. Dacă într-o clasă există multe câmpuri, trebuie să ne întrebăm dacă toate acestea trebuie să fie accesibile din exterior. Nu are sens ca toate câmpurile private ale unei clase să fie expuse în afară, chiar şi prin intermediul proprietăților.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

7. Aplicaţii

7.1. Realizaţi interfaţa grafică a unei aplicaţii Windows pentru rezolvarea de ecuaţii polinomiale de gradul I şi II, precum şi de ecuaţii trigonometrice simple (figura 2.5).

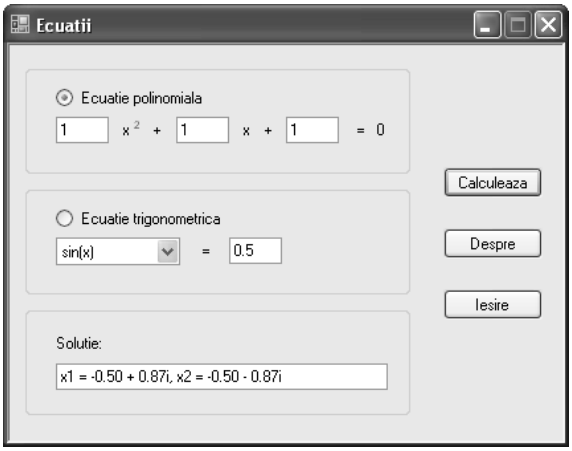


Figura 2.5. *Exemplu de rezolvare*

7.2. Creaţi clasele pentru rezolvarea celor două tipuri de ecuaţii (figura 2.6). Se vor arunca două tipuri de excepţii, conform structurii de mai jos. Fiind două tipuri diferite de ecuaţii, vom avea o interfaţă IEquation cu metoda Solve, din care vor fi derivate PolyEquation şi TrigEquation.

În clasa principală, evenimentul de tratare a apăsării butonului *Calculeaza* va testa tipul de ecuaţie.

Pentru o ecuaţie polinomială, va citi coeficienţii ecuaţiei şi va instanţia un obiect corespunzător:

```
IEquation eq = new PolyEquation(x2, x1, x0);
textBoxSolutie.Text = eq.Solve();
```

Pentru o ecuaţie trigonometrică, depinzând de tipul de funcţie trigonometrică:

```
eq = new TrigEquation(TrigEquation.TrigonometricFunction.Sin, arg);
textBoxSolutie.Text = eq.Solve();
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

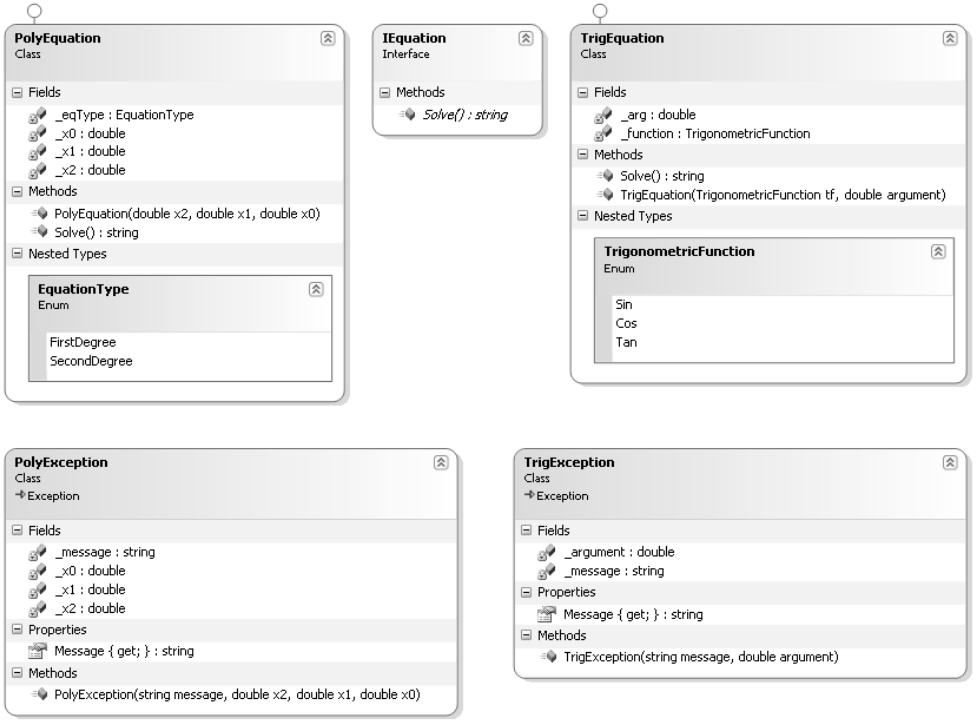


Figura 2.6. *Exemplu de rezolvare: clasele soluției*

Fiecărui tip de ecuație îi va corespunde propria clasă de excepție: PolyException, respectiv TrigException.

Pentru ecuația polinomială se vor considera următoarele excepții:

- pentru $x_2 = x_1 = x_0 = 0$: „O infinitate de soluții”;
- pentru $x_2 = x_1 = 0$ și $x_0 \neq 0$: „Nicio soluție”.

Pentru ecuația trigonometrică, excepția „Argument invalid” va fi aruncată când $|arg| > 1$, însă numai pentru funcțiile *arcsin* și *arccos*.

În evenimentul butonului vom avea un singur bloc try, urmat de trei blocuri catch: pentru cele două tipuri de excepție, precum și pentru excepțiile generice, rezultate de exemplu la încercarea de a citi coeficienții, dacă utilizatorul introduce caractere nenumerice.

Indicație: codul pentru rezolvarea ecuației polinomiale este prezentat în continuare, unde $\Delta = x_1^2 - 4 \cdot x_2 \cdot x_0$:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
if (_x2 == 0)
{
        _eqType = EquationType.FirstDegree;
        // soluţie: -_x0 / _x1
}
else if (delta > 0)
{
        double sqrtDelta = Math.Sqrt(delta);
        double sol1 = (-_x1 + sqrtDelta) / (2.0 * _x2);
        double sol2 = (-_x1 - sqrtDelta) / (2.0 * _x2);
        // soluţii: sol1, sol2
}
else if (delta == 0)
{
        double sol = (-_x1) / (2.0 * _x2);
        // soluţie: sol
}
else
{
        double rsol = -_x1 / (2.0 * _x2);
        double isol = Math.Sqrt(-delta) / (2.0 * _x2);
        // soluţii: rsol ± isol
}
```

Atenţie: scopul laboratorului este lucrul cu excepții, nu rezolvarea propriu-zisă a ecuațiilor!

7.3. Eliminați blocurile try-catch din clasa principală a aplicației şi efectuați tratarea excepțiilor pe firul de execuție în metoda Main.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 3

Reutilizarea codului cu ajutorul DLL-urilor

1. Obiective
2. Bibliotecile legate dinamic
3. Crearea DLL-urilor în C#
4. Grafică în C#
5. Aplicaţii

1. Obiective

Obiectivele capitolului 3 sunt următoarele:

- Descrierea DLL-urilor;
- Legarea statică a DLL-urilor .NET, cea mai simplă şi folosită formă;
- Legarea dinamică a DLL-urilor .NET, utilă pentru crearea *plug-in*-urilor;
- Prezentarea unor aspecte privind grafica în C#.

2. Bibliotecile legate dinamic

Bibliotecile legate dinamic (engl. "Dynamic Link Libraries", DLL), reprezintă implementarea Microsoft a conceptului de *bibliotecă partajată* (engl. "shared library"). În trecut, DLL-urile au dat dezvoltatorilor posibilitatea de a crea biblioteci de funcţii care puteau fi folosite de mai multe aplicaţii. Însuşi sistemul de operare Windows a fost proiectat pe baza DLL-urilor. În timp ce avantajele modulelor de cod comun au extins oportunităţile dezvoltatorilor, au apărut de asemenea probleme referitoare la actualizări şi revizii. Dacă un program se baza pe o anumită versiune a unui DLL şi alt program actualiza acelaşi DLL, de multe ori primul program înceta să mai funcţioneze corect.

Pe lângă problemele legate de versiuni, dacă se dorea dezinstalarea unei aplicaţii, se putea şterge foarte uşor un DLL care era încă folosit de un alt program.

Soluţia propusă de Microsoft a fost introducerea posibilităţii de a urmări folosirea DLL-urilor cu ajutorul registrului, începând cu Windows

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

95. Se permitea unei singure versiuni de DLL să ruleze în memorie la un moment dat. Când era instalată o nouă aplicație care folosea un DLL existent, se incrementa un contor. La dezinstalare, contorul era decrementat şi dacă nicio aplicație nu mai folosea DLL-ul, atunci acesta putea fi şters.

Apare însă o altă problemă deoarece când un DLL este încărcat, Windows va folosi versiunea ce rulează până când nicio aplicație nu o mai foloseşte. Astfel, chiar dacă DLL-ul sistemului este în regulă, sau o aplicație are o copie locală pe care se lucrează, când aplicația *precedentă* a pornit cu o versiune incompatibilă, atunci noua aplicație nu va merge.

Această problemă se poate manifesta în situații precum următoarele: o aplicație nu funcționează atunci când rulează o altă aplicație sau, şi mai ciudat, o aplicație nu funcționează dacă o altă aplicație a rulat (dar nu mai rulează neapărat în prezent). Dacă aplicația *A* încarcă o bibliotecă incompatibilă sau coruptă, atunci aplicația *B* lansată foloseşte această bibliotecă. Această versiune va sta în memorie chiar după ce aplicația *A* nu mai există (atât timp cât aplicația *B* încă rulează), deci aplicația *B* s-ar putea să înceteze să funcționeze din cauza aplicației *A*, chiar dacă aceasta nu mai rulează. O a treia aplicație *C* poate să eşueze (câtă vreme aplicația *B* încă rulează) chiar dacă este pornită după ce aplicația *A* a fost închisă.

Rezultatul s-a numit *infernul DLL* (engl. "DLL hell").

Rezolvarea infernului DLL a fost unul din scopurile platformei .NET. Aici pot exista mai multe versiuni ale unui DLL ce rulează simultan, ceea ce permite dezvoltatorilor să adauge o versiune care funcționează la programul lor fără să se îngrijoreze că un alt program va fi afectat. Modul în care .NET reuşeşte să facă aceasta este prin renunțarea la folosirea registrului pentru a lega DLL-urile de aplicații şi prin introducerea conceptului de *assembly*.

3. Crearea DLL-urilor în C#

Din fereastra corespunzătoare *File → New → Project*, se alege tipul proiectului *Class Library* (figura 3.1). În *namespace*-ul proiectului pot fi adăugate mai multe clase. În acest caz, din exterior fiecare clasă va fi accesată ca `Namespace.ClassName`. Dacă se elimină *namespace*-ul, clasa va fi accesată direct cu numele clasei: `ClassName`. Spre deosebire de o aplicație executabilă, aici nu va exista o metodă `Main`, deoarece DLL-ul este numai o bibliotecă de funcții utilizabile din alte programe executabile.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

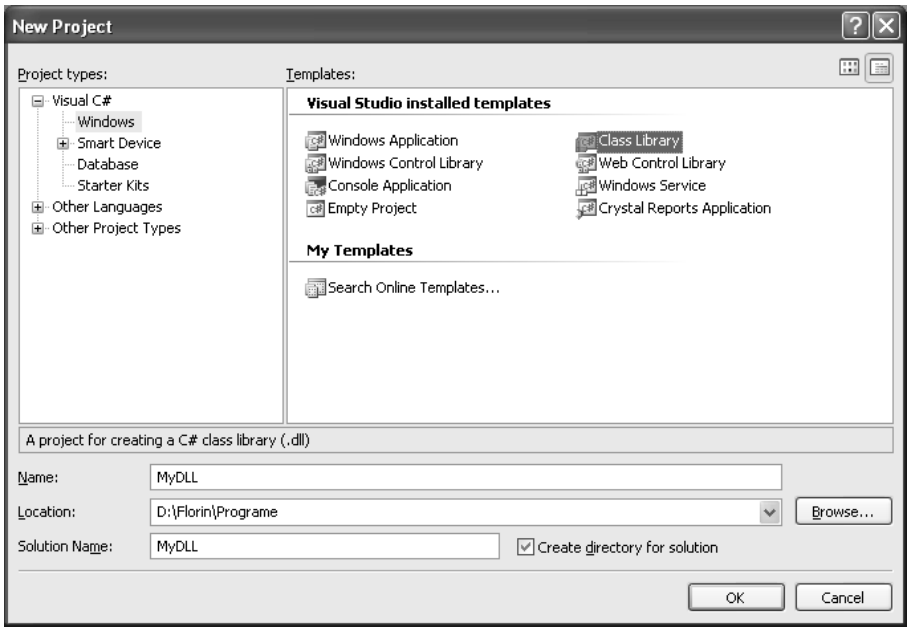


Figura 3.1. *Crearea unui proiect de tip Class Library (DLL)*

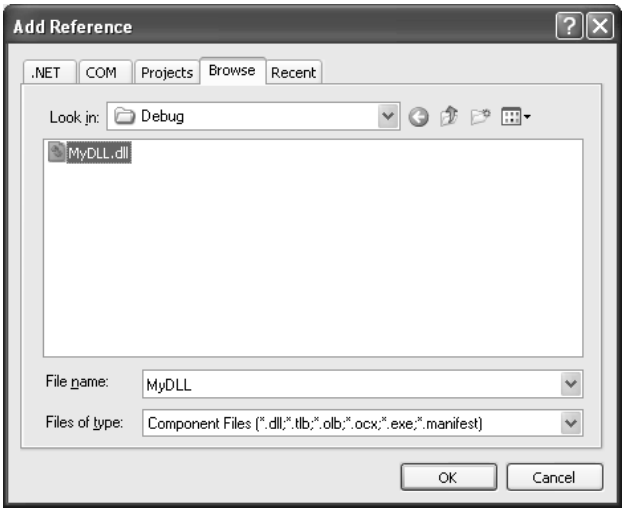


Figura 3.2. *Adăugarea unei referinţe la un DLL*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

3.1. Legarea statică

După ce s-a creat un proiect corespunzător unei aplicații executabile, din *Project → Add Reference*, se selectează de pe harddisk în *tab-page*-ul *Browse* fișierul DLL care trebuie adăugat în proiect. În continuare, în program vor fi utilizate ca atare toate funcțiile din DLL (figura 3.2).

Să considerăm următorul exemplu: avem într-un DLL numit Operatii.dll o clasă Putere în *namespace*-ul Matematica având o metodă double Patrat(double x) care returnează valoarea parametrului ridicată la pătrat. Mai întâi, se va căuta şi selecta fişierul Operatii.dll de pe harddisk. Apoi, într-o metodă din clasa programului principal, *se va apela direct metoda* de ridicare la putere:

```
double a = Matematica.Putere.Patrat(5.5);
```

3.2. Legarea dinamică

Legarea statică presupune că se cunoaşte ce DLL va trebui încărcat înainte de executarea programului. Există totuşi situații în care acest lucru este imposibil: de exemplu, dacă o aplicație necesită o serie de *plug-in*-uri, acestea pot fi adăugate sau şterse, iar aplicația principală trebuie să determine după lansarea în execuție cu ce DLL-uri poate lucra. Un alt avantaj este faptul că programatorul poate testa existența unui anumit DLL necesar şi poate afişa un mesaj de eroare şi eventual o modalitate de corectare a acesteia.

C# permite încărcarea dinamică a DLL-urilor. Să considerăm tot exemplul anterior. Apelul metodei de ridicare la pătrat se face în modul următor:

```
// se încearcă încărcarea DLL-ului
Assembly a = Assembly.Load("Operatii");

// se identifică tipul (clasa) care trebuie instanţiată
// dacă în clasa din DLL există un namespace,
// se foloseşte numele complet al clasei din assembly, incluzând namespace-ul
Type t = a.GetType("Matematica.Putere");

// se identifică metoda care ne interesează
MethodInfo mi = t.GetMethod("Patrat");
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
// se creează o instanţă a clasei dorite
// aici se apelează constructorul implicit
object o = Activator.CreateInstance(t);

// definim un vector de argumente pentru a fi trimis metodei
// metoda Pătrat are numai un argument de tip double
object[] args = new object[1];
double x = 5.5;
args[0] = x;

// apelul efectiv al metodei şi memorarea rezultatului
double result = (double)mi.Invoke(o, args);
```

Acesta este modul general de încărcare. Este obligatorie *tratarea excepțiilor* care pot apărea datorită unei eventuale absențe a DLL-ului sau a încărcării incorecte a unei metode. De aceea, fluxul de mai sus va trebui împărțit în mai multe blocuri care să permită tratarea excepțiilor, *încărcarea o singură dată* a bibliotecii şi *apelarea de câte ori este nevoie* a metodelor dorite.

Dacă se apelează dinamic o metodă statică, se foloseşte null în loc de obiectul o.

La începutul programului, dacă avem o aplicație Windows, în evenimentul Load al ferestrei trebuie să existe un bloc de tipul:

```
private void Form1_Load(object sender, EventArgs e)
{
    try
    {
        LoadOperatiiPutere();
    }
    catch (Exception exc)
    {
        MessageBox.Show(exc.Message, "Exceptie DLL");
        Close();
    }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Metoda care încarcă efectiv DLL-ul este:

```
private void LoadOperatiiPutere()
{
    Type t = null;

    try
    {
        Assembly a = Assembly.Load("Operatii");
        t = a.GetType("Matematica.Putere");
    }
    catch (Exception)
    {
        throw new Exception("Operatii.dll nu poate fi incarcat");
    }

    _miPatrat = t.GetMethod("Patrat");
    if (_miPatrat == null)
        throw new Exception("Metoda Patrat din Operatii.dll nu poate fi accesata");

    _objPutere = Activator.CreateInstance(t);
}
```

Metoda clasei din programul executabil care realizează apelul efectiv este:

```
private double MyOperatiiPutere(double x)
{
    object[] args = new object[1];
    args[0] = x;
    return (double)_miPatrat.Invoke(_objPutere, args);
}
```

Pentru compilarea codului de mai sus este necesară includerea în program a *namespace*-ului System.Reflection.

3.3. Depanarea unui DLL

Deoarece un DLL nu este direct executabil, există două metode pentru dezvoltarea şi depanarea unei astfel de componente.

Abordarea cea mai simplă este crearea unei aplicații executabile, de cele mai multe ori de tip consolă în cazul în care DLL-ul va conține funcții

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

de calcul şi nu grafice. În proiect va exista *namespace*-ul DLL-ului cu toate clasele aferente, iar în plus o clasă cu o metodă Main, din care se vor putea apela şi depana metodele din DLL. La sfârşit, după ce DLL-ul este corect, se poate exclude clasa cu Main şi se poate recompila proiectul ca DLL: *Project Properties → Application → Output type*. Alternativ, se poate crea un nou proiect de tip *Class library* în care se va copia codul DLL-ului.

A doua metodă este includerea a două proiecte într-o soluție în Visual Studio. Un proiect va fi DLL-ul iar celălalt proiect va fi unul executabil. Din proiectul executabil trebuie adăugată referința la DLL. În consecință, fişierul *dll* rezultat din compilare va fi copiat automat în directorul fişierului *exe*.

4. Grafică în C#

Pentru lucrul în mod grafic, C# pune la dispoziția programatorului o clasă numită Graphics. Pentru a transla suprafața de desenare în cadrul ferestrei, se poate folosi un obiect de tip PictureBox, care va fi plasat acolo unde se doreşte.

Important! Desenarea în fereastră, într-un PictureBox sau în orice alt control trebuie făcută numai în evenimentul Paint al controlului respectiv. În caz contrar, când fereastra este minimizată sau când controlul este acoperit de alte ferestre, desenul se pierde.

Evenimentul Paint conține un parametru de tipul:

```
System.Windows.Forms.PaintEventArgs e
```

Suprafața grafică a controlului va fi în acest caz e.Graphics, care conține metodele de desenare. Majoritatea controalelor au implementat un eveniment Paint. În afara acestuia, suprafața de desenare poate fi identificată prin metoda CreateGraphics, care returnează un obiect de tip Graphics. În cele ce urmează, vom aminti unele elemente de bază, care pot fi folosite în program ca atare (totuşi, pentru aprofundarea acestor cunoştințe, se pot consulta alte manuale sau documentația *MSDN*). Deoarece majoritatea metodelor sunt supraîncărcate, vom da câte un exemplu simplu de utilizare pentru fiecare situație.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Desenarea unei linii

```
e.Graphics.DrawLine(Pen pen, int x1, int y1, int x2, int y2)
```

Primul parametru precizează culoarea şi stilul de desenare a liniei. Se poate declara şi folosi un nou obiect de tip Pen (Pen pen = new Pen(...)), însă pentru linii continue şi culori predefinite se poate utiliza enumerarea Pens. De exemplu:

```
e.Graphics.DrawLine(Pens.Black, 10, 10, 20, 40);
```

Desenarea unui dreptunghi şi a unei elipse

```
e.Graphics.DrawRectangle(Pen pen, int x, int y, int latime, int inaltime)
```

Desenează conturul unui dreptunghi, cu un anumit stil şi culoare, folosind coordonatele unui vârf şi respectiv lăţimea şi înălţimea sa. Pentru umplerea unui dreptunghi cu un anumit model şi culoare, se foloseşte metoda:

```
e.Graphics.FillRectangle(Brush b, int x, int y, int latime, int inaltime)
```

Ca şi în cazul unui Pen, se poate declara un nou obiect de tip Brush, de exemplu:

```
Brush b = new SolidBrush(Color.Blue);
```

sau se poate folosi enumerarea Brushes, care presupune că stilul de umplere va fi solid (compact).

Un dreptunghi alb cu contur negru se va desena astfel:

```
e.Graphics.FillRectangle(Brushes.White, 0, 0, 10, 20);
e.Graphics.DrawRectangle(Pens.Black, 0, 0, 10, 20);
```

În mod analog se folosesc metodele DrawEllipse şi FillEllipse.

Afişarea unui text în mod grafic

```
DrawString (string s, Font font, Brush brush, int x, int y)
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Primul parametru este textul care trebuie afişat. Al doilea parametru reprezintă corpul de literă cu care se va desena şirul de caractere. De exemplu:

```
Font font = new Font("Arial", 10);
```

Al treilea parametru este utilizat pentru trasarea efectivă a textului (vezi paragraful despre desenarea unui dreptunghi şi a unei elipse). Ultimii doi parametri sunt coordonatele ecran.

Double-buffering automat

Deoarece funcţiile grafice sunt în general lente, pentru desene suficient de complexe sau animaţii, unde se presupune ştergerea unor elemente şi afişarea altora, ecranul pare să clipească (engl. "flickering"). O soluţie este desenarea tuturor elementelor într-o zonă de memorie, de exemplu un Bitmap, şi apoi afişarea directă a acestuia pe ecran. Această afişare presupune de obicei doar copierea unor informaţii dintr-o zonă de memorie în alta, fiind deci foarte rapidă. Deoarece se folosesc două zone de memorie, această tehnică se numeşte *double-buffering*. În anumite situaţii, se pot folosi mai multe zone de memorie.

În C# se poate face automat *double-buffering*, prin introducerea unei linii precum următoarea (de obicei, dar nu obligatoriu) în constructorul ferestrei:

```
SetStyle(ControlStyles.AllPaintingInWmPaint | ControlStyles.DoubleBuffer |
   ControlStyles.UserPaint, true);
```

Versiunea 2.0 a platformei .NET (cu Visual Studio 2005) a introdus proprietatea DoubleBuffered pentru obiectele Form.

5. Aplicaţii

5.1. Realizaţi un DLL numit Prim care să conţină o clasă cu o metodă care testează dacă un număr natural, primit ca parametru, este prim. *Notă:* 1 nu este număr prim.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

5.2. Realizați un program executabil care testează conjectura lui Goldbach: orice număr par mai mare sau egal ca 4 poate fi scris ca sumă de 2 numere prime şi orice număr impar mai mare sau egal ca 7 poate fi scris ca sumă de 3 numere prime (figura 3.3). Va fi folosită funcția de test pentru numere prime din *Prim.dll*. Legarea se va face *static*.

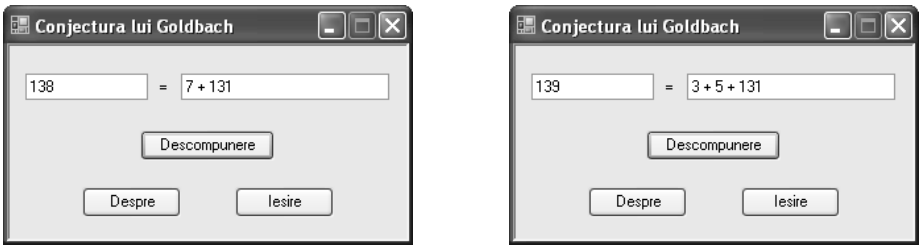


Figura 3.3. *Exemplu de rezolvare*

5.3. Modificați *Prim.dll* prin adăugarea unei metode `int NumaraPrime(int n)`, care calculează numărul numerelor prime mai mici sau egale cu n. Verificați că *Suma.exe* se execută corect după modificarea DLL-ului.

5.4. Realizați un program executabil care încarcă *dinamic* biblioteca *Prim.dll* şi apelează metoda `NumaraPrime`.

5.5. *Temă pentru acasă*. Realizați un program executabil care să afişeze graficul funcției:

$$f(n) = \text{numărul de numere prime} \leq n, n > 0.$$

precum şi o aproximare a acestui număr, de forma:

$$g(n) = \frac{n}{\log n},\ n > 2.$$

Pentru calculul exact, se va utiliza metoda `int NumaraPrime(int n)` din *Prim.dll*. Pentru aproximare, se va crea un DLL numit *Aproximare*, cu o clasă cu acelaşi nume care să conțină o metodă `double XLogX(double x)`.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Legarea se va face *dinamic*. Verificați tratarea excepțiilor în cazul utilizării unui DLL cu o semnătură incorectă a metodei XLogX.

Indicații. Pentru fiecare metodă din DLL-uri, este utilă definirea unei metode corespunzătoare în programul principal. De exemplu, pentru metoda EstePrim din *Prim.dll*, se poate crea o metodă de tipul:

```
private bool MyPrimEstePrim(int n)
```

care să apeleze metoda EstePrim și să returneze rezultatul. În continuare, în program se va apela direct metoda MyPrimEstePrim. Încărcarea DLL-ului trebuie făcută *o singură dată*, la pornirea programului, chiar dacă apelul metodelor se va face ori de câte ori este necesar.

Cele două funcții nu se suprapun, doar forma lor este asemănătoare. De aceea, cele două grafice trebuie scalate diferit pe axa *Y* (figura 3.4).

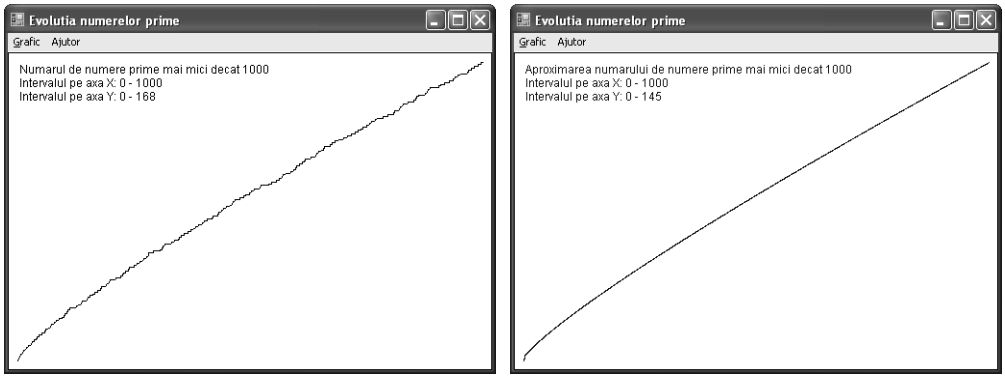


Figura 3.4. *Exemplu de rezolvare*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 4

Documentarea unui proiect. Fişiere de ajutor

1. Obiective
2. Crearea de fişiere de ajutor
3. Activarea unui fişier de ajutor prin program
4. Generarea automată a documentaţiei *API*
5. Comentariile
6. Lucrul cu fişiere în C#: încărcare, salvare
7. Aplicaţii

1. Obiective

Obiectivele principale ale capitolului 4 sunt următoarele:

- Crearea de fişiere de ajutor (*help*) în formatele HLP şi CHM;
- Activarea unui fişier de ajutor dintr-un program C#;
- Generarea automată a documentaţiei *API* a unei soluţii C# cu ajutorul utilitarului *NDoc*.

În plus, ne propunem:

- Să oferim o serie de recomandări privind comentarea unui program;
- Să prezentăm utilizarea dialogurilor de încărcare şi salvare, precum şi o modalitate simplă de a lucra cu fişiere în C#;
- Să utilizăm o clasă care poate rezolva pătrate magice de orice dimensiune (atât de dimensiune impară cât şi de dimensiune pară).

2. Crearea de fişiere de ajutor

2.1. Crearea de fişiere HLP

Un utilitar gratuit pentru realizarea de fişiere HLP este *Microsoft Help Workshop* (figura 4.1). Acesta creează fişiere HLP pe baza unui

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

document *Rich Text Format*, RTF, editat după anumite convenții, corespunzătoare opțiunilor fișierelor de ajutor.

Subiectele din *help* sunt asociate în general cu un identificator unic. Acesta se inserează printr-o notă de subsol (engl. "footnote") cu caracterul **#** înaintea titlului paginii respective. Deschiderea unei anumite pagini de ajutor, atât din fișierul „cuprins", cât și dintr-un program, se face pe baza acestui identificator. Paginile sunt despărțite cu separator de pagină (engl. "page break").

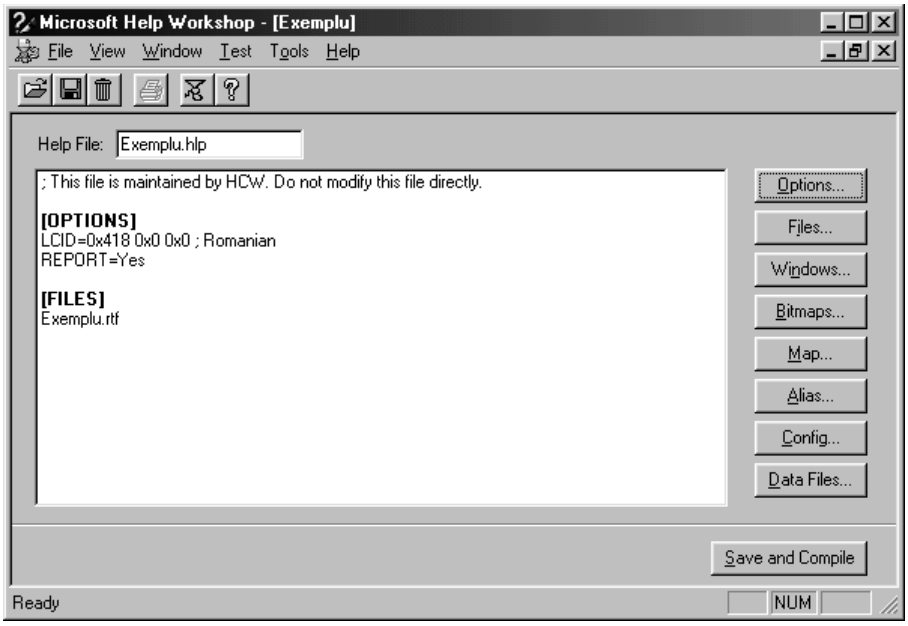


Figura 4.1. *Microsoft Help Workshop*

Fișierul de ajutor propriu-zis poate fi însoțit de un fișier „cuprins", cu formatul următor:

```
:Base Exemplu.hlp
:Title Exemplu de fisier .hlp
:Index=Exemplu.hlp
1 Capitolul 1
2 Pagina 1=Topic_id1
2 Pagina 2=Topic_id2
1 Capitolul 2
2 Pagina 3=Topic_id3
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Base reprezintă numele fişierului de ajutor, titlul determină textul ce va apărea pe bara ferestrei *help*-ului, iar indexul precizează fişierul de unde se va face indexarea (în cazul nostru, acelaşi fişier). În continuare, se descrie structura *help*-ului într-o manieră arborescentă. Numerele din faţa denumirilor de capitole reprezintă nivelul în arbore al subiectului respectiv (figura 4.2).

Figura 4.2. *Cuprinsul unui fişier HLP*

Se observă că legătura la paginile corespunzătoare se face pe baza identificatorului de subiect, *topic id*.

Pentru ca utilizatorul să navigheze uşor prin *help*, sunt disponibile opţiuni de indexare şi căutare a cuvintelor cheie. În cazul indexului, cuvintele cheie sunt desemnate printr-o notă de subsol marcată **K**. Pagina de index afişează lista cuvintelor cheie definite pentru fiecare subiect (figura 4.3).

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Figura 4.3. *Indexul unui fişier HLP*

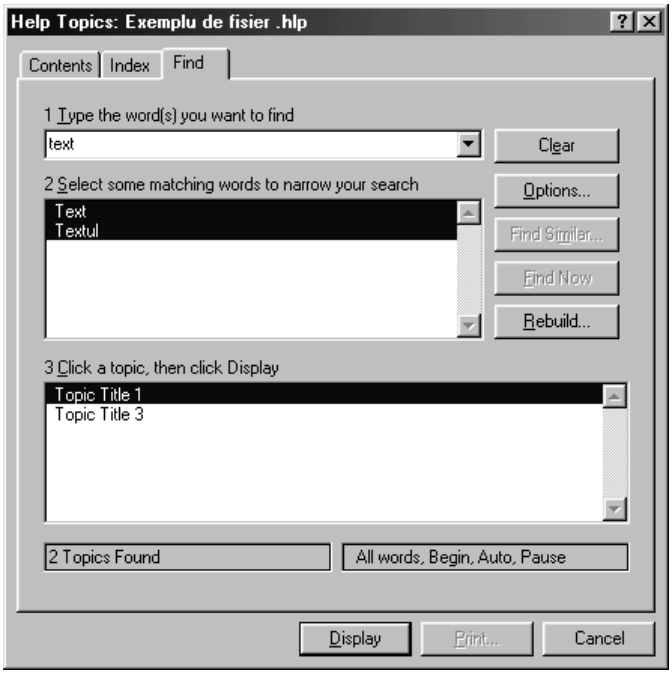


Figura 4.4. *Pagina de căutare într-un fişier* HLP

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

În pagina de căutare *Find* (figura 4.4), *help*-ul generează automat o listă cu toate cuvintele găsite. Utilizatorul poate introduce un anumit cuvânt, sau mai multe, şi află în ce pagini apare acesta. Titlurile de subiecte care apar în lista de jos sunt determinate de o notă de subsol marcată cu **$** în fişierul RTF.

Dacă se doreşte includerea unor imagini, acestea sunt pur şi simplu inserate în fişierul RTF şi vor apărea în mod automat şi în *help*.

O altă opțiune utilă este includerea de texte *pop-up*, în situații în care explicarea unui termen sau a unui concept este suficient de scurtă şi nu necesită utilizarea unei pagini noi (figura 4.5).

Figura 4.5. *Text pop-up într-un fişier* HLP

Acest format presupune inserarea unui text „ascuns" în fişierul RTF. Dacă editorul folosit este Microsoft Word, atunci trebuie să activăm mai întâi opțiunea de vizualizare a informațiilor ascunse, prin combinația de taste *CTRL+** sau *CTRL+SHIFT+8*. Textul *link* va fi subliniat şi imediat după el va fi introdus un identificator pentru fereastra mică ce va apărea. Identificatorul va fi scris cu litere ascunse, ceea ce se poate realiza din meniul *Format → Font… → Hidden*. Să presupunem că identificatorul se numeşte *POPUP*.

Într-o altă pagină se va scrie textul care se doreşte să apară (în cazul nostru: *„Textul apare într-o fereastră mică"*). În fața sa, va fi inserată o notă

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

de subsol marcată cu **#,** iar conținutul notei va fi identificatorul menționat anterior, *POPUP*.

Pentru realizarea unei legături la altă pagină, se va sublinia *dublu* textul corespunzător legăturii, care va fi urmat de identificatorul subiectului paginii la care se vrea să se sară.

2.2. Crearea de fişiere CHM

Un utilitar gratuit pentru realizarea de fişiere CHM (*Compiled HTML*) este *HTML Help Workshop* (figura 4.6).

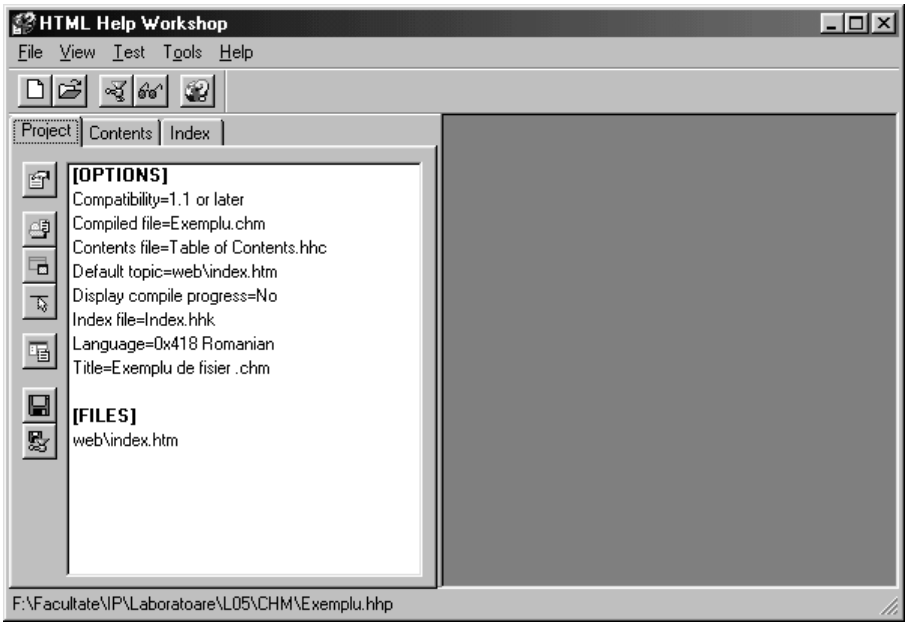


Figura 4.6. *Microsoft HTML Help Workshop*

Ideea care stă la baza acestui format este transformarea unui site web sau a unui grup de pagini HTML într-un singur fişier, cu opțiuni de navigare şi căutare.

Pentru a realiza un astfel de fişier, trebuie create mai întâi paginile HTML cu informațiile utile. În *tab-page*-ul *Project* se apasă al doilea buton din stânga, *Add/Remove topic files*. Este suficientă includerea paginii de index, de la care se presupune că există legături către celelalte pagini. Se creează apoi câte un fişier *Contents* şi *Index*.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

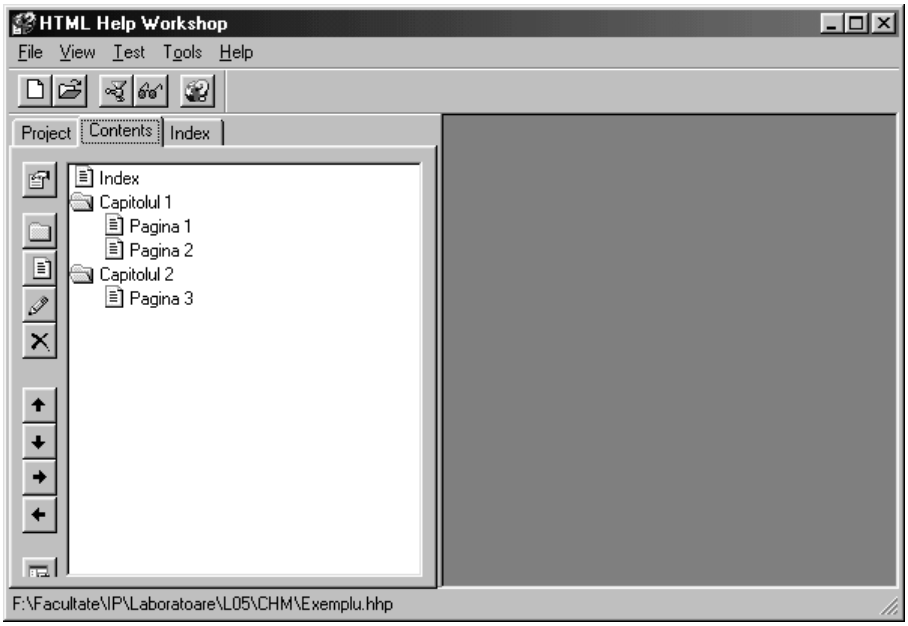


Figura 4.7. *Crearea cuprinsului unui fişier* CHM

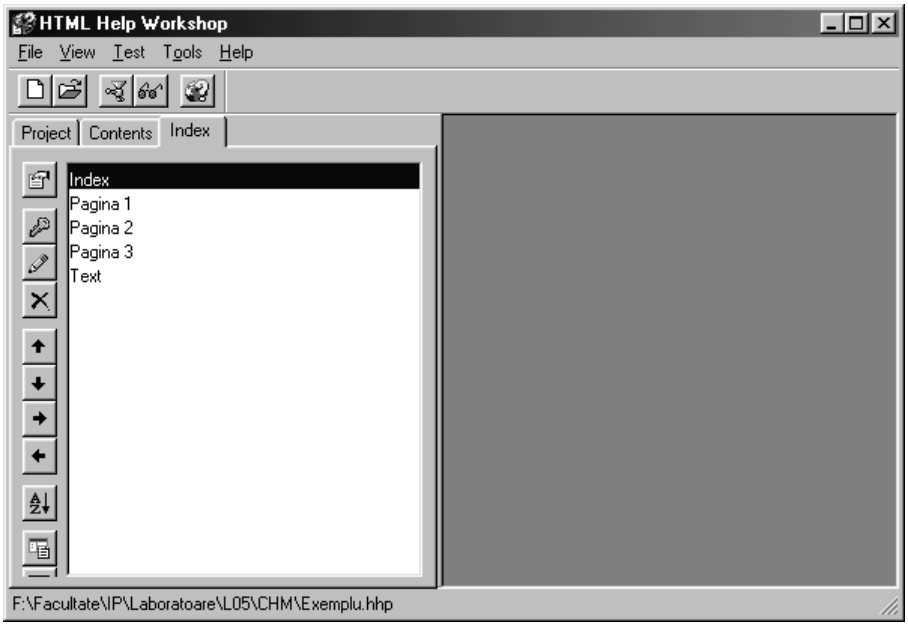


Figura 4.8. *Crearea intrărilor de index ale unui fişier* CHM

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

În *tab-page*-ul *Contents* (figura 4.7), se pot insera subiectele corespunzătoare unor anumite pagini. Pentru aceasta se folosesc butoanele din stânga *Insert a heading* (un nod în arbore) şi *Insert a page* (o frunză).

În mod analog se definesc şi intrări de index, care pot fi asociate cu una sau mai multe pagini (figura 4.8).

Dacă o intrare de index are mai multe pagini asociate, la căutare rezultatul va fi de forma celui prezentat în figura 4.9.

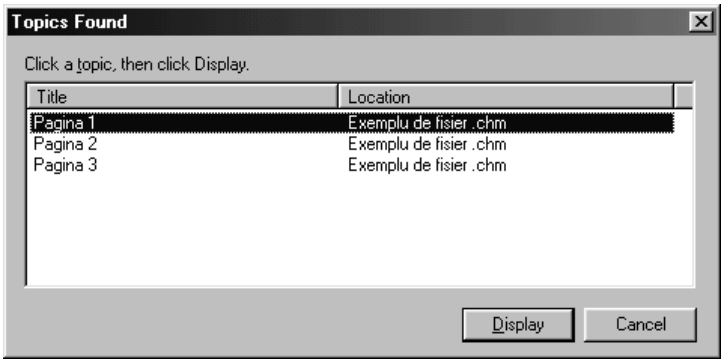


Figura 4.9. *Rezultatele căutării într-un fişier* CHM

Pentru generarea automată a opțiunii de căutare în lista de cuvinte a paginilor, se apasă primul buton din stânga din *tab-page*-ul *Project*, numit *Change project options*, iar în pagina *Compiler* se bifează căsuța *Compile full-text search information*.

3. Activarea unui fişier de ajutor prin program

3.1. *Process.Start*

Cel mai simplu mod de deschidere a unui fişier de ajutor este printr-un apel la sistemul de operare. În C# apelul este de forma:

```
System.Diagnostics.Process.Start("nume-fisier");
```

3.2. *HelpProvider*

O altă modalitate este utilizarea clasei specializate HelpProvider. Se introduce în Form un astfel de obiect şi apoi din fereastra de proprietăți se

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

setează numele fişierului CHM asociat în câmpul HelpNamespace. Desigur această operație şi cele descrise în continuare pot fi făcute şi prin program. Apoi, pentru fiecare control din fereastră pentru care dorim să apară *help*-ul atunci când apăsăm tasta F1, trebuie să modificăm următoarele proprietăți:

- Show help on help provider: true;
- Help navigator on help provider: locul unde vrem să se deschidă *help*-ul: la pagina de cuprins, la pagina de index, de căutare sau la o pagină specificată de programator;
- Help keyword on help provider: dacă pentru controlul respectiv avem o anumită pagină care trebuie deschisă, Help navigator on help provider va lua valoarea Topic iar în Help keyword on help provider se va introduce calea către pagina care trebuie deschisă, relativ la fişierul CHM. De exemplu, dacă fişierul este obținut prin compilarea unui director numit *web*, în care se găseşte un document *pag1.htm*, care trebuie deschis acum, în acest câmp se va introduce: *web\pag1.htm*.

3.3. *Help*

Pentru activarea unui fişier de ajutor CHM se poate folosi şi clasa Help. Aceasta are un număr de metode statice specifice, precum ShowHelp, ShowHelpIndex, ShowPopup.

Pentru acelaşi exemplu de fişier CHM vom avea:

- Help.ShowHelp(this, "Exemplu.chm")
  - deschide fişierul *Exemplu.chm*;
- Help.ShowHelp(this, "Exemplu.chm", "web/pag1.htm")
  - deschide pagina solicitată din acelaşi fişier;
- Help.ShowHelpIndex(this,"Exemplu.chm")
  - deschide pagina de index a fişierului *Exemplu.chm*;
- Help.ShowPopup(this, "Pop-up window",
  new Point(Cursor.Position.X, Cursor.Position.Y))
  - deschide o fereastră de *pop-up* cu textul dorit la coordonatele curente ale *mouse*-ului.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

4. Generarea automată a documentaţiei *API*

NDoc generează documentaţie pentru Interfaţa de Programare a Aplicaţiei (engl. "Application Programming Interface", API) pe baza *assembly*-urilor .NET şi a fişierelor de documentaţie XML generate de compilatorul C#. *NDoc* poate genera documentaţia în diferite formate, precum *help*-ul HTML în stil MSDN, CHM, formatul Visual Studio .NET (*HTML Help 2*), sau stilul de pagini web de tip *MSDN online*.

Documentaţia XML se realizează automat prin includerea în codul sursă a comentariilor triple: linii care încep cu /// şi care preced un tip definit de utilizator cum ar fi: *namespace*-uri, clase, interfeţe sau membrii precum câmpuri, metode, proprietăţi sau evenimente.

Pentru a genera automat fişierul XML, trebuie setată calea către acesta în *Project Properties → Build → Output → XML documentation file* (figura 4.10).

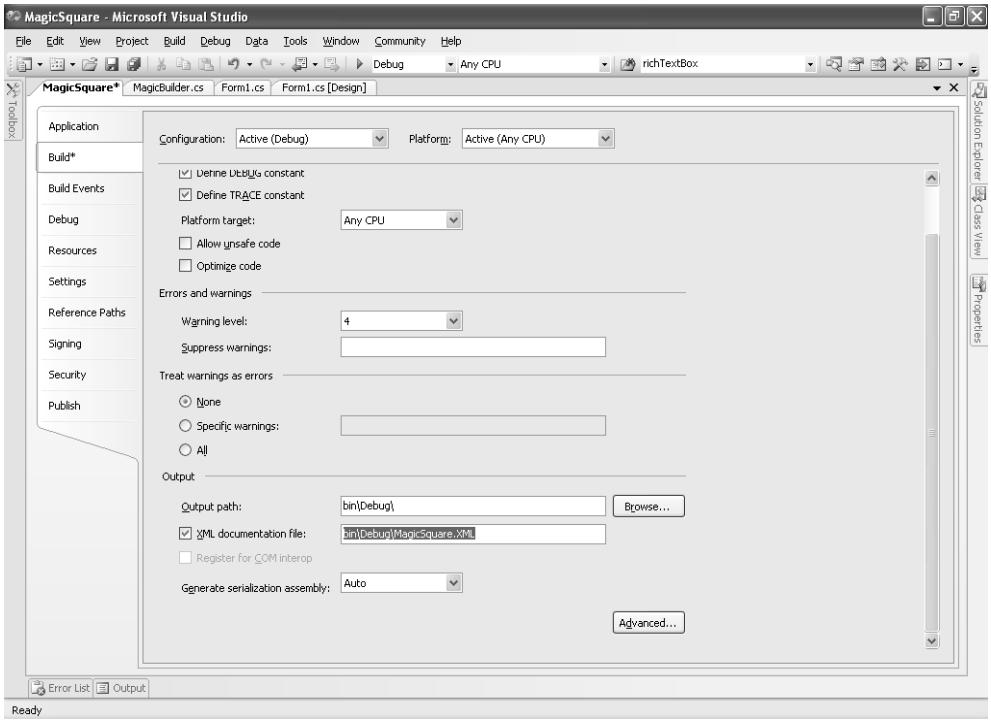


Figura 4.10. *Generarea automată a unui fişier de documentaţie XML*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

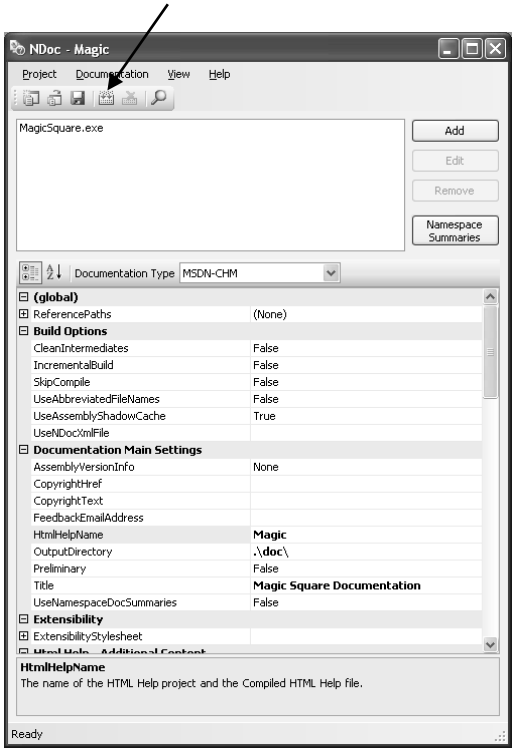


Figura 4.11. *Utilitarul NDoc*

În programul *NDoc*, se adaugă *assembly*-ul pentru care se va crea documentaţia şi se setează proprietăţile proiectului – ce anume va fi inclus în documentaţie. De exemplu, o *API* pentru un utilizator extern nu va documenta metodele private, deoarece acestea oricum nu vor putea fi accesate de utilizator. Pentru a genera o referinţă de uz intern, se pot include şi aceste metode prin setarea proprietăţii respective.

Tot aici se alege titlul documentaţiei şi formatul acesteia (*Documentation type*). Apoi se apasă butonul *Build Docs* din bara de instrumente (figura 4.11).

Va rezulta documentaţia în formatul ales (figura 4.12).

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

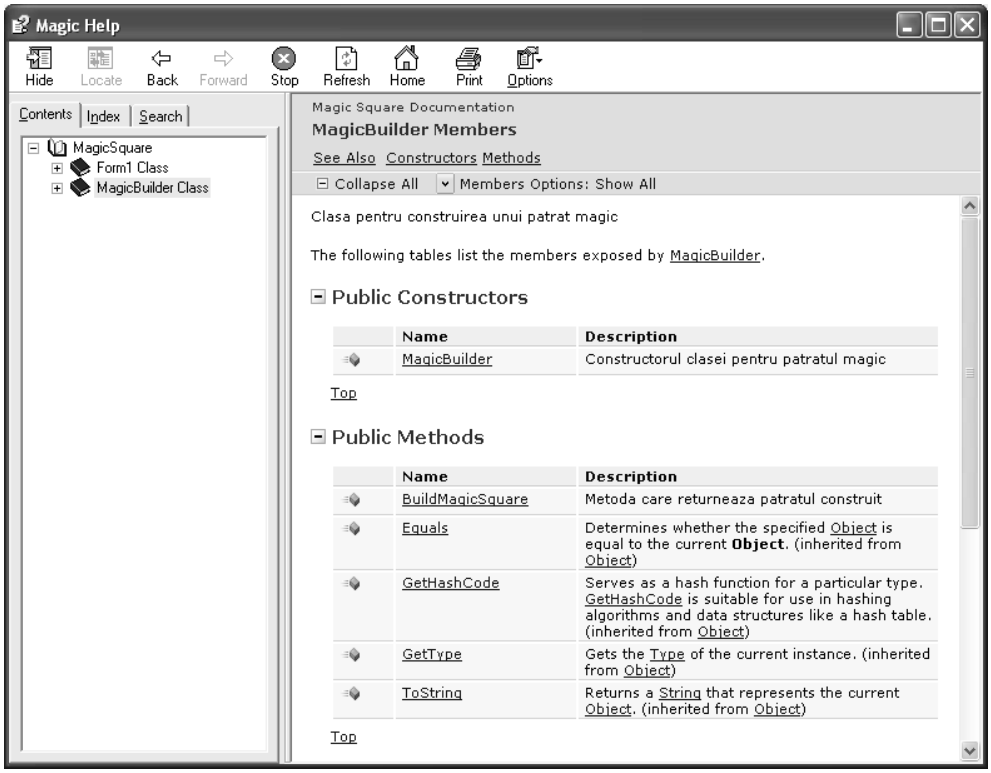


Figura 4.12. *Documentaţie API creată cu NDoc*

5. Comentariile

În cele ce urmează, se prezintă câțiva paşi de bază pentru a îmbunătăți calitatea comentariilor.

Trebuie să se explice *DE CE*, nu *CUM.* Comentariile nu trebuie să descrie cum lucrează programul; se poate vedea aceasta citind codul, care trebuie scris clar şi inteligibil. Trebuie în schimb să ne concentrăm asupra explicării motivelor pentru care s-a scris în acest fel sau ce îndeplineşte în final un bloc de instrucțiuni.

Trebuie verificat dacă se scrie constant *actualizarea structurii StudentList din StRegistry* sau *memorarea informaţiilor despre obiectele Student pentru a fi folosite mai târziu*. Cele două formulări sunt echivalente, dar a doua precizează scopul codului pe când prima spune doar ce face codul. În timpul întreținerii acelei părți din cod, motivele pentru care aceasta există se vor schimba mult mai rar decât modalitățile concrete de

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

implementare. Întreținerea celui de-al doilea tip de comentarii este deci mult mai uşoară. Comentariile bune explică de ce, nu cum.

De asemenea, se pot folosi comentarii pentru a explica alegerea unei anumite implementări. Dacă există două strategii de implementare posibile şi se optează asupra uneia din ele, atunci se pot adăuga comentarii pentru a explica motivele alegerii.

Nu trebuie descris codul. Comentariile descriptive inutile sunt evidente: ++i; *// incrementează i*. Unele pot să fie mult mai subtile: comentarii descriptive lungi ale unui algoritm complex, urmat de implementarea algoritmului. Nu este nevoie să se rescrie laborios codul în limbaj natural decât dacă se documentează un algoritm complex care ar fi altfel impenetrabil. Nu trebuie duplicat codul în comentarii.

Nu trebuie înlocuit codul. Dacă există comentarii care specifică ceva ce s-ar putea aplica prin însuşi limbajul de programare, de exemplu *această variabilă ar trebui accesată doar de către clasa A*, atunci acest deziderat trebuie exprimat printr-o sintaxă concretă. Dacă ne găsim în situația în care scriem comentarii pentru a explica cum lucrează un algoritm complex, atunci trebuie să ne oprim. Este bine să documentăm codul, dar ar fi şi mai bine dacă am putea face codul sau algoritmul mai clar:

- Dacă se poate, codul trebuie împărțit în câteva funcții bine denumite pentru a reflecta logica algoritmului;
- Nu trebuie scrise comentarii care să descrie folosirea unei variabile; aceasta trebuie redenumită. Comentariile pe care vrem să le scriem ne spun deseori care ar trebui să fie numele variabilei;
- Dacă se documentează o condiție care ar trebui să fie întotdeauna îndeplinită, poate ar trebui să se scrie o aserțiune de testare a unităților (mai multe detalii se vor vedea în capitolele 13 şi 14);
- Nu este nevoie de optimizări premature care pot obscuriza codul.

Când ne aflăm în situația de a scrie comentarii dense pentru a explica codul, trebuie sa facem un pas înapoi, întrucât s-ar putea să existe o problemă mai mare care trebuie rezolvată.

Codul neaşteptat trebuie documentat. Dacă o parte din cod este neobişnuită, neaşteptată sau surprinzătoare, trebuie documentată cu un comentariu. Ne va fi mult mai uşor mai târziu când vom reveni, uitând totul

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

despre problemă. Dacă există soluții alternative specifice (*workarounds*), acestea trebuie menționate în comentarii.

Comentariile trebuie să fie clare. Comentariile se folosesc pentru a adnota şi explica codul. Acestea nu trebuie sa fie ambigue, ci din contra, cât mai specifice. Dacă o persoană citeşte comentariile şi rămâne să se întrebe ce înseamnă, atunci acestea au scăzut calitatea programului şi au afectat înțelegerea codului.

Comentariile ajută la citirea codului. Comentariile sunt de obicei scrise deasupra codului pe care-l descriu, nu dedesubt. În acest fel, codul sursă se citeşte în jos, aproape ca o carte. Comentariile ajută la pregătirea cititorului pentru ceea ce urmează să vină. Folosite cu spații verticale, comentariile ajută la împărțirea codului în „paragrafe". Un comentariu introduce câteva linii, explicând ce se intenționează să se obțină, Urmează imediat codul, apoi o linie goală, apoi următorul bloc. Există o convenție: un comentariu cu o linie goală înaintea lui apare ca un început de paragraf, în timp ce un comentariu intercalat între două linii de cod apare mai mult ca o propoziție în paranteze sau o notă de subsol.

Comentariile din antetul fişierului. Fiecare fişier sursă ar trebui să înceapă cu un bloc de comentarii ce descrie conținutul său. Acesta este doar o scurtă prezentare, o prefață, furnizând câteva informații esențiale ce se doresc întotdeauna afişate de îndată ce este deschis un fişier. Dacă există acest antet, atunci un programator care deschide fişierul va avea încredere în conținut; arată că fişierul a fost creat aşa cum trebuie. Funcționalitatea fiecărui fişier sursă trebuie comentată.

Unele persoane susțin că antetul ar trebui să furnizeze o listă cu toate funcțiile, clasele, variabilele globale şi aşa mai departe, care sunt definite în fişier. Acesta este un dezastru pentru întreținere; un astfel de comentariu devine rapid învechit. Antetul fişierului trebuie să conțină informații despre scopul fişierului (de exemplu *implementarea interfeței IDocument*) şi o declarație cu drepturile de autor care să descrie proprietarul şi regulile de copiere.

Antet-ul *nu* trebuie să conțină informații care ar putea deveni uşor învechite, precum data când a fost ultima oară modificat fişierul. Probabil că data nu ar fi actualizată des şi ar induce în eroare. De asemenea, nu trebuie să conțină un istoric al fişierului sursă care să descrie toate modificările făcute. Dacă trebuie să derulezi peste 10 pagini din istoricul modificărilor pentru a ajunge la prima linie de cod, atunci devine incomod lucrul cu fişierul. Din acest motiv, unii programatori pun o astfel de listă la sfârşitul

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

fişierului. Chiar şi aşa, acesta poate deveni foarte mare şi se va încărca mai încet.

Degradarea comentariilor. Orice cod neîntreţinut corect tinde să se degradeze, pierzând din calitatea proiectării iniţiale. Totuşi, comentariile tind să se degradeze mult mai repede decât oricare altă parte de cod. Ele îşi pot pierde sincronizarea cu codul pe care îl descriu şi pot deveni profund supărătoare. Comentariile incorecte sunt mai dăunătoare decât lipsa comentariilor: dezinformează şi induc în eroare cititorul.

Cea mai simplă soluţie este aceasta: când se repară, adaugă sau modifică codul, se repară, adaugă sau modifică orice comentarii din jurul său. Nu se modifică câteva linii şi atât. Trebuie să ne asigurăm că orice modificare din cod nu le va transforma în comentarii neadevărate. Corolarul este următorul: trebuie să creăm comentarii uşor de actualizat, dacă nu, ele nu vor fi actualizate. Comentariile trebuie să fie clar legate de secţiunea lor de cod şi nu trebuie plasate în locaţii obscure.

6. Lucrul cu fişiere în C#: încărcare, salvare

Clasele OpenFileDialog şi SaveFileDialog afişează dialoguri de încărcare/salvare a fişierelor. Aceste obiecte trebuie apelate din alte componente, de exemplu, la apăsarea unui buton sau la alegerea unei opţiuni dintr-un meniu, va apărea ferestra de dialog. În funcţia apelantă va trebui introdus un bloc de tipul:

```
if (openFileDialog.ShowDialog() != DialogResult.OK)  // nu s-a apăsat OK
    return;
```

Metoda de mai sus determină afişarea dialogului. Dacă aceasta se execută corect (utilizatorul a ales un fişier), este disponibilă proprietatea *open/saveFileDialog*.FileName, care conţine numele fişierului dorit (cale completă şi nume).

Câteva proprietăţi:

- *open/saveFileDialog.*DefaultExt – extensia ataşată în mod automat fişierului;
- *open/saveFileDialog.*Filter – dialogul de selecţie de fişiere include un *combo-box* cu tipurile fişierelor. Când utilizatorul alege un tip de fişier din listă, numai fişierele de tipul selectat sunt afişate în dialog.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Filter poate fi setat în *Properties* sau în codul sursă, în formatul: "Text Files (*.txt)|*.txt|All Files (*.*)|*.*";

- *open/saveFileDialog.*InitialDir – directorul implicit unde se deschide dialogul. Poate fi de exemplu *MyDocuments*, dacă această proprietate nu este specificată. Pentru directorul în care se află programul, se foloseşte „ . ”.

În continuare, se vor defini nişte *stream*-uri pentru fişiere. De exemplu:

```
StreamWriter sw = new StreamWriter(saveFileDialog.FileName);
// operaţii cu sw
// de exemplu scriem în fişier un număr n cu 3 zecimale
sw.WriteLine("Numarul este {0:F3}", n);
sw.Close();
```

Pentru lucrul cu fişiere trebuie inclus *namespace*-ul System.IO.

7. Aplicaţii

7.1. Realizați o interfață grafică pentru desenarea unui pătrat magic (figura 4.13), cu ajutorul clasei MagicBuilder, prezentată în continuare.

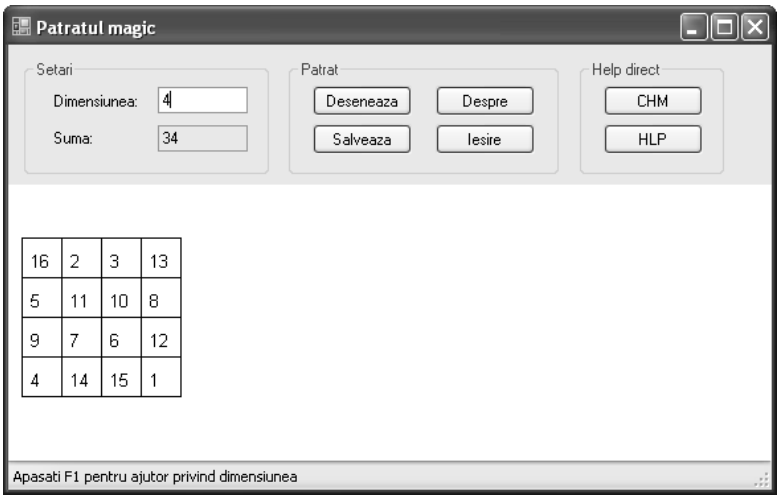


Figura 4.13. *Exemplu de rezolvare*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
/************************************************************************
 *                                                                      *
 *  File:        MagicBuilder.cs                                        *
 *  Copyright:   (c) 2003, A. Riazi                                     *
 *  Website:     http://www.codeproject.com/KB/recipes/Magic_Square.asp  *
 *  Description: Calculates magic squares of any size.                  *
 *               Translated into C# and adapted by Florin Leon          *
 *               http://florinleon.byethost24.com/lab_ip.htm            *
 *                                                                      *
 *  This code and information is provided "as is" without warranty of   *
 *  any kind, either expressed or implied, including but not limited    *
 *  to the implied warranties of merchantability or fitness for a       *
 *  particular purpose. You are free to use this source code in your    *
 *  applications as long as the original copyright notice is included.  *
 *                                                                      *
 ************************************************************************/

using System;
using System.Collections.Generic;

namespace MagicSquare
{
   public class MagicBuilder
   {
      private int[,] _matrix;
      private int _size;

      public MagicBuilder(int size)
      {
         _size = size;
         _matrix = new int[size, size];
      }

      public int[,] BuildMagicSquare()
      {
         if (_size < 1 || _matrix == null)
            throw new Exception("Dimensiune incorecta");
         MagicSquare(_matrix, _size);
         return _matrix;
      }

      private void MagicSquare(int[,] matrix, int n)
      {
         if (n % 2 == 1)
            OddMagicSquare(matrix, n);
         else
         {
            if (n % 4 == 0)
               DoublyEvenMagicSquare(matrix, n);
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
        else
            SinglyEvenMagicSquare(matrix, n);
    }
}

private void OddMagicSquare(int[,] matrix, int n)
{
    int nsqr = n * n;
    int i = 0, j = n / 2;

    for (int k = 1; k <= nsqr; ++k)
    {
        matrix[i, j] = k;

        i--;
        j++;

        if (k % n == 0)
        {
            i += 2;
            --j;
        }
        else
        {
            if (j == n)
                j -= n;
            else if (i < 0)
                i += n;
        }
    }
}

private void DoublyEvenMagicSquare(int[,] matrix, int n)
{
    int[,] mat1 = new int[n, n];
    int[,] mat2 = new int[n, n];

    int i, j;

    int index = 1;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {
            mat1[i, j] = ((i + 1) % 4) / 2;
            mat2[j, i] = ((i + 1) % 4) / 2;
            matrix[i, j] = index;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
            index++;
        }

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        {
            if (mat1[i, j] == mat2[i, j])
                matrix[i, j] = n * n + 1 - matrix[i, j];
        }
}

private void SinglyEvenMagicSquare(int[,] matrix, int n)
{
    int p = n / 2;

    int[,] mat = new int[p, p];
    MagicSquare(mat, p);

    int i, j, k;

    for (i = 0; i < p; i++)
        for (j = 0; j < p; j++)
        {
            matrix[i, j] = mat[i, j];
            matrix[i + p, j] = mat[i, j] + 3 * p * p;
            matrix[i, j + p] = mat[i, j] + 2 * p * p;
            matrix[i + p, j + p] = mat[i, j] + p * p;
        }

    if (n == 2)
        return;

    int[] mat1 = new int[p];
    List<int> vect = new List<int>();

    for (i = 0; i < p; i++)
        mat1[i] = i + 1;

    k = (n - 2) / 4;

    for (i = 1; i <= k; i++)
        vect.Add(i);

    for (i = n - k + 2; i <= n; i++)
        vect.Add(i);
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
        int temp;
        for (i = 1; i <= p; i++)
          for (j = 1; j <= vect.Count; j++)
          {
            temp = matrix[i - 1, vect[j - 1] - 1];
            matrix[i - 1, vect[j - 1] - 1] = matrix[i + p - 1, vect[j - 1] - 1];
            matrix[i + p - 1, vect[j - 1] - 1] = temp;
          }

        i = k;
        j = 0;
        temp = matrix[i, j]; matrix[i, j] = matrix[i + p, j]; matrix[i + p, j] = temp;

        j = i;
        temp = matrix[i + p, j]; matrix[i + p, j] = matrix[i, j]; matrix[i, j] = temp;
      }
    }
}
```

Indicații: Un pătrat magic este o matrice pătratică de dimensiune n, care conține numerele întregi din intervalul $1 - n^2$ și în care suma elementelor pe linii, coloane și diagonale este aceeași.

Pentru salvarea graficului, trebuie să se folosească un obiect de tip Bitmap, care dispune de funcții de salvare/încărcare a imaginilor. Deoarece obiectul Bitmap va fi folosit atât pentru desenarea într-un PictureBox cât și pentru salvarea imaginii într-un eveniment de apăsare a unui buton, va trebui să avem un câmp în clasă, instanțiat în constructorul ferestrei:

```
private Bitmap _bmp;

public void MainForm()
{
   InitializeComponent();

   // în constructorul ferestrei, după metoda InitializeComponent();
   _bmp = new Bitmap(pictureBox.Width, pictureBox.Height);
}
```

Desenarea în Bitmap trebuie pusă în legătură cu suprafața de desenare a ferestrei din PictureBox. Atunci, în evenimentul Paint al acesteia, vom avea o secvență de cod de forma:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
Graphics g = Graphics.FromImage(_bmp);

// în continuare, desenarea se va face în g (în Bitmap)
g.Clear(Color.White);
...

// la sfârşit, vom desena conţinutul bitmap-ului în picture-box
e.Graphics.DrawImage(_bmp, 0, 0);
```

Salvarea imaginii din Bitmap se va face în evenimentul Click al unui buton:

```
_bmp.Save(saveFileDialog.FileName, System.Drawing.Imaging.ImageFormat.Png);
```

Cu linia de cod de mai sus, imaginea va fi salvată în format *png*.

7.2. Creaţi un fişier HLP sau un fişier CHM pentru programul *Pătratul magic*.

Indicaţie: Cu ajutorul componentei HelpProvider se poate afişa un *help* bazat pe context, de exemplu pentru *text-box*-urile corespunzătoare dimensiunii pătratului şi sumei caracteristice.

7.3. Creaţi o documentaţie *API* pentru clasa MagicBuilder folosind programul *NDoc*. Comentariile triple trebuie introduse cu o linie mai sus de prima linie a clasei, câmpului sau metodei comentate.

Exemple:

```
/// <summary>
/// Clasa pentru construirea unui pătrat magic
/// </summary>
public class MagicBuilder
...

    /// <summary>
    /// Constructorul clasei pentru pătratul magic
    /// </summary>
    /// <param name="size">Dimensiunea pătratului</param>
    public MagicBuilder(int size)
    ...
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
/// <summary>
/// Metoda care returnează pătratul construit
/// </summary>
/// <returns>Matricea corespunzătoare pătratului magic</returns>
public int[,] BuildMagicSquare()
...
```

7.4. *Opțional.* Realizați un program de generare a unui antet cu informații pentru un fișier de cod sursă C# (figurile 4.14 și 4.15).

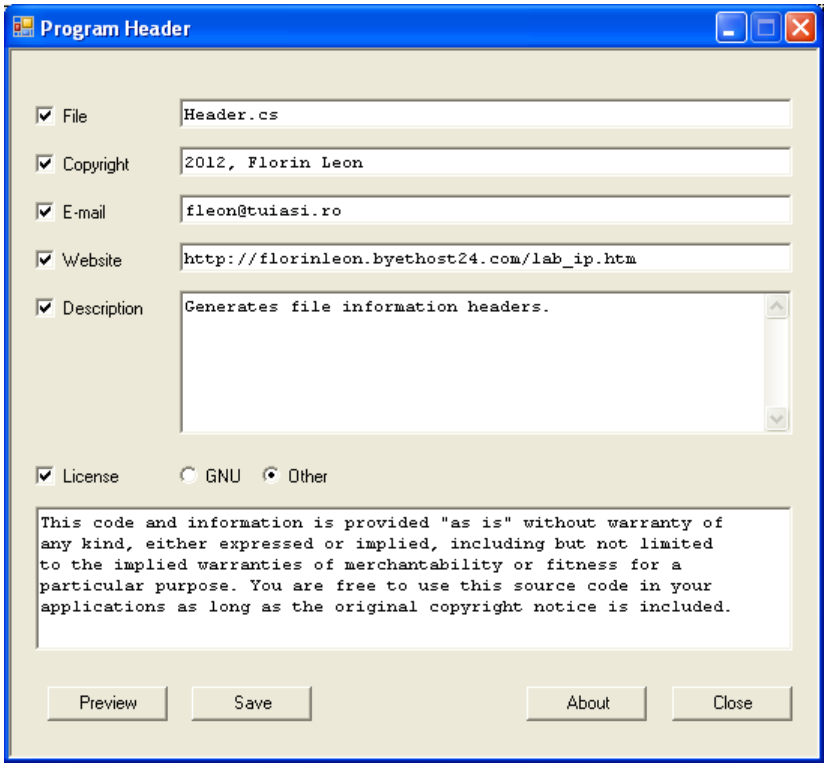


Figura 4.14. *Program generator de antete*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

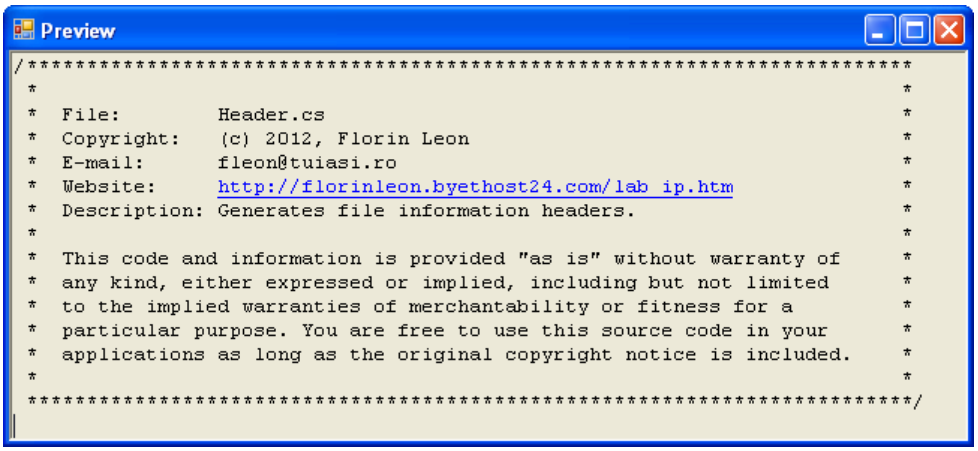


Figura 4.15. *Antet generat pentru un fişier sursă*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 5

Diagrame UML

1. Obiective
2. Diagrame principale ale UML
3. *Altova UModel*
4. Aplicaţii

1. Obiective

Limbajul Unificat de Modelare (engl. "Unified Modeling Language", UML) este un limbaj pentru specificarea, vizualizarea, construirea şi documentarea elementelor sistemelor software. Este un standard *de facto* pentru modelarea software. Obiectivele capitolului 5 sunt următoarele:

1. Prezentarea celor mai importante tipuri de diagrame UML 2.0;
2. Introducerea programului *Altova UModel* pentru desenarea diagramelor UML:
    a. Utilizarea diagramei de clase pentru generarea automată de cod C#;
    b. Generarea automată a diagramei de clase pe baza codului sursă C#;
    c. Desenarea unor diagrame de cazuri de utilizare, clase, activităţi şi secvenţe.

2. Diagrame principale ale UML

2.1. Diagrama cazurilor de utilizare

O diagramă de nivel înalt utilă în multe situaţii este diagrama cazurilor de utilizare, care descrie mulţimea de interacţiuni dintre utilizator şi sistem. Prin construirea unei colecţii de cazuri de utilizare, putem descrie întregul sistem într-o manieră clară şi concisă. Cazurile de utilizare sunt denumite de obicei printr-o combinaţie verb-substantiv, de exemplu:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Plăteşte factura, Creează cont etc. Notația pentru un caz de utilizare este prezentată în figura 5.1.

Figura 5.1. *Caz de utilizare*

Un caz de utilizare trebuie să aibă un inițiator al acțiunii, numit *actor*. În cazul unui sistem bancar, retragerea banilor este făcută de clienți, astfel încât clientul devine unul din actori (figura 5.2).

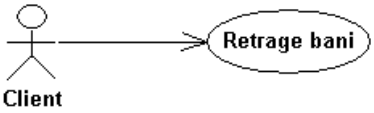


Figura 5.2. *Caz de utilizare cu actor*

Actorii nu sunt numai oameni, ci orice cauză externă care inițiază un caz de utilizare, de exemplu un alt sistem de calcul sau un concept mai abstract, precum timpul, de exemplu în ultima zi a lunii se actualizează statele de salarii. Pentru majoritatea sistemelor, un anumit actor poate interacționa cu mai multe cazuri de utilizare, iar un anumit caz de utilizare poate fi inițiat de actori diferiți (figura 5.3).

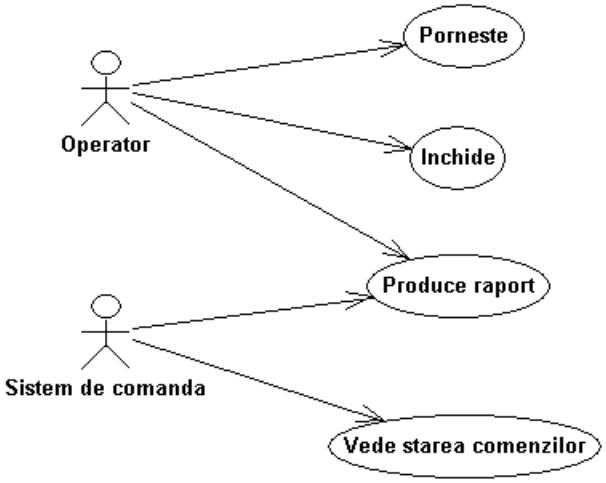


Figura 5.3. *Cazuri de utilizare cu actori multipli*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

2.2. Diagrama de clase

Modelarea conceptuală (numită şi „modelarea domeniului") este activitatea de identificare a conceptelor importante pentru sistem. În cazul proiectării orientate obiect, modelarea conceptuală se realizează prin diagrama claselor, întrucât clasele reprezintă concepte. Diagrama claselor furnizează structura codului care urmează să fie scris. Problema principală este identificarea conceptelor. Regula de urmat aici este: *dacă clientul nu înțelege conceptul, probabil că nu este un concept*.

O clasă se reprezintă printr-o căsuță împărțită în trei (figura 5.4). În partea de sus este notat numele clasei, în partea mediană sunt incluse atributele (câmpurile) iar în partea de jos operațiile (metodele) sale.

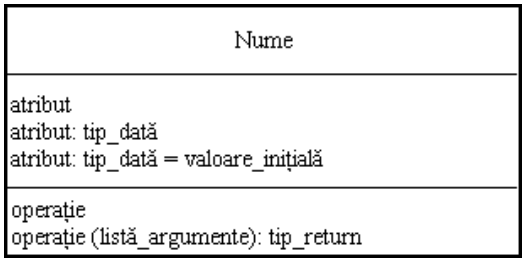


Figura 5.4. *O clasă în notaţia UML*

În figura 5.5 sunt prezentate notațiile pentru vizibilitatea atributelor şi operațiilor (private, protejate, publice).

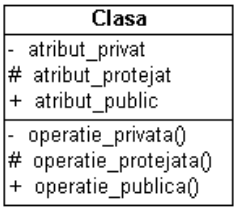


Figura 5.5. *Vizibilitatea atributelor şi operaţiilor*

2.2.1. Dependenţa

Relația de dependență apare când o clasă foloseşte pentru scurt timp o altă clasă, de exemplu trimiterea unui mesaj (apelarea din clasa `A` a unei metode din clasa `B`) sau trimiterea ca parametru într-o metodă a clasei `A` a

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

unui obiect de tip B. În C#, un exemplu este clasa Math, ale cărei metode statice sunt apelate punctual de obiectele altor clase. Se notează cu linie punctată cu o săgeată (figura 5.6).

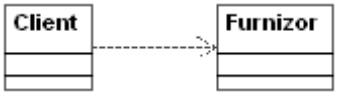


Figura 5.6. *Relația de dependență*

2.2.2. Asocierea

Linia simplă în UML are rolul de asociere: o clasă A are un câmp instanțiat din cealaltă clasă B. Numerele descriu cardinalitatea asocierii, adică ne spun câte instanțe sunt permise din fiecare clasă. Figura 5.7 prezintă câteva cardinalități posibile, deși din punct de vedere al notației nu există restricții asupra cardinalităților care pot fi specificate.

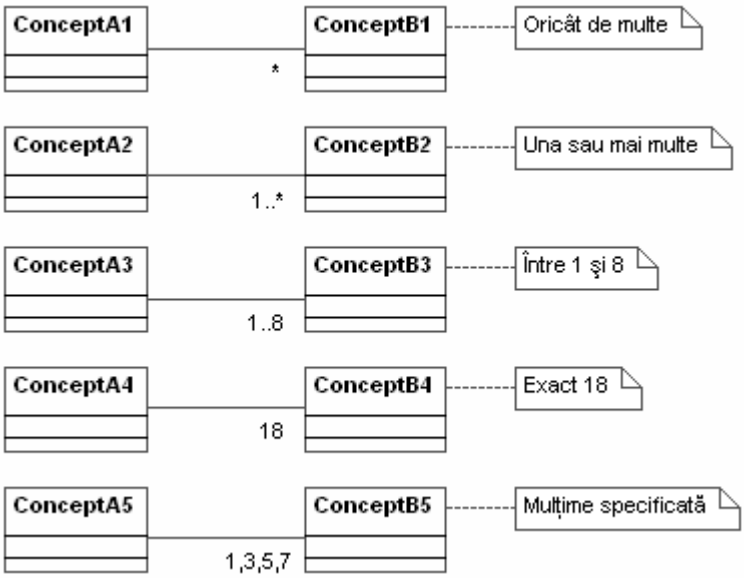


Figura 5.7. *Cardinalităţi*

O greşeală pe care o putem face în faza de analiză este să trasăm o linie între două clase, dar să nu notăm numele asocierii. După ce vom trasa toate liniile, nu vom mai şti ce înseamnă fiecare.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

În figura 5.8 este prezentat un exemplu de asociere între clase, corespunzător fazei de analiză.

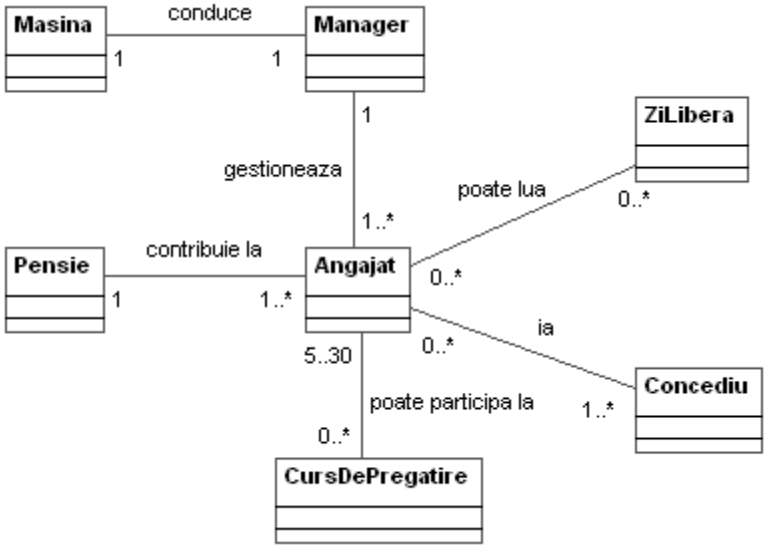


Figura 5.8. *Asociere mai complexă*

Asocierile de mai sus sunt bidirecționale. Într-o asociere unidirecțională, cele două clase sunt înrudite, dar numai o clasă ştie că există relația respectivă. În situația din figura 5.9, managerul ştie despre adresă, dar adresa nu ştie despre manager.

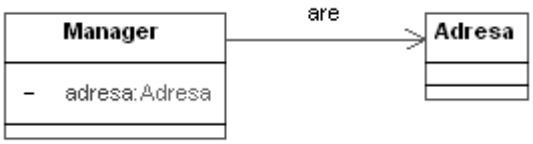


Figura 5.9. *Asociere unidirecţională*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

2.2.3. Agregarea şi compunerea

În cazul agregării, un obiect este construit din altele. De exemplu, un calculator este o agregare între procesor, placă video, placă de sunet etc. (figura 5.10).

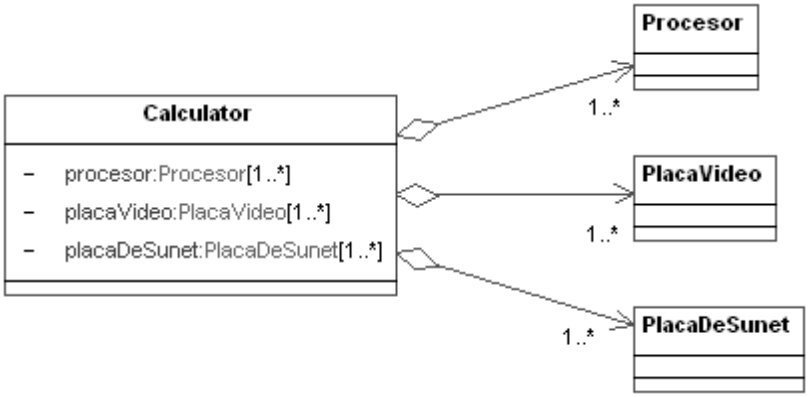


Figura 5.10. *Relaţia de agregare*

Compunerea este un concept similar cu agregarea, însă mai puternic, deoarece implică faptul că întregul nu poate exista fără părți. În exemplul de agregare de mai sus, dacă se înlătură placa de sunet, calculatorul rămâne calculator. Însă o carte nu poate exista fără pagini; o carte este *compusă* din pagini. Notația este asemănătoare, dar rombul este plin (figura 5.11).

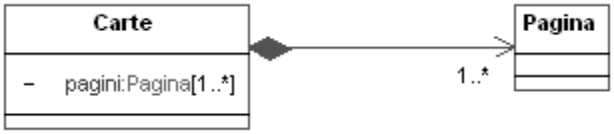


Figura 5.11. *Relaţia de compunere*

2.2.4. Moştenirea

De multe ori, mai multe clase au atribute şi operaţii comune. Acestea pot fi introduse într-o singură clasă şi moştenite în celelalte, de exemplu clasele din figura 5.12.

Dacă am mai vrea să adăugăm o clasă Pisică, ar trebui să repetăm atributele şi operaţiile comune. Soluția este moştenirea dintr-o clasă mai generală.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

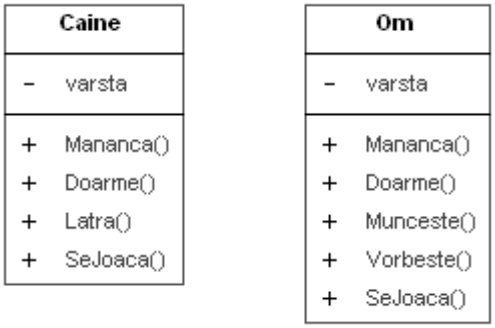


Figura 5.12. *Clase cu potenţial de generalizare*

Notația UML pentru moştenire (generalizare) este cea din figura 5.13.

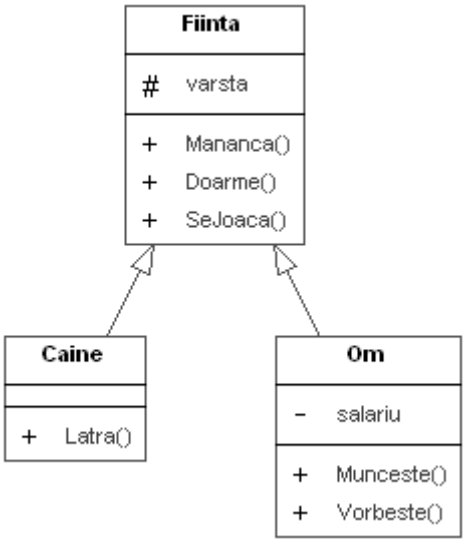


Figura 5.13. *Relaţia de moştenire*

Trebuie să subliniem faptul că atributul vârstă a fost transformat din privat în protejat, pentru a putea fi accesat în clasele derivate.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

2.2.5. Metode abstracte şi virtuale

Clasele derivate pot redefini implementarea unor metode. În exemplul următor, clasele Lup, Peşte şi AltAnimal sunt derivate din clasa Animal, însă primele două reimplementează metoda Mănâncă în modul lor specific. Notația în acest caz este cea din figura 5.14.

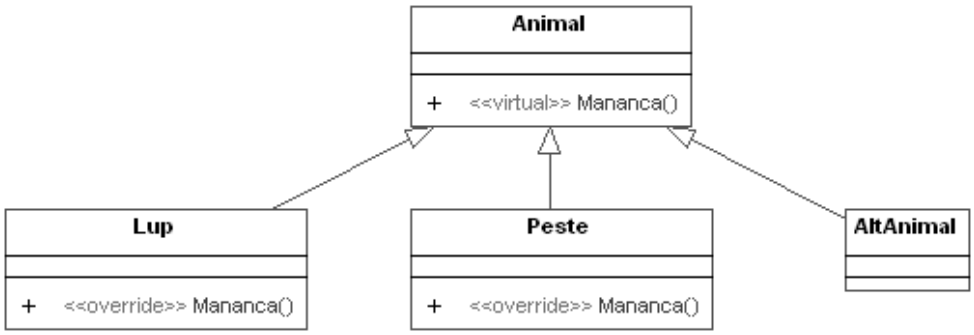


Figura 5.14. *Notaţia pentru metode virtuale*

Cuvintele introduse între „<<” şi „>>” se numesc *stereotipuri*.

De multe ori avem nevoie să lăsăm o metodă neimplementată într-o clasă (metodă abstractă) şi să o implementăm pe un nivel mai de jos al ierarhiei. Clasele şi metodele abstracte se notează cu italice (figura 5.15).

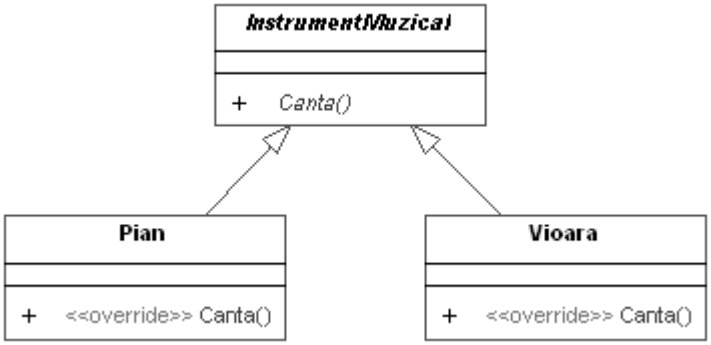


Figura 5.15. *Notaţia pentru clase şi metode abstracte*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

2.2.6. Interfeţe

Să presupunem că InstrumentMuzical din exemplul precedent e acum o interfață iar clasele Pian şi Vioară trebuie să implementeze metoda Cântă. Notația este asemănătoare celei de la moştenirea de clase, dar cu linie punctată, iar interfața poate fi declarată explicit cu un stereotip (figura 5.16).

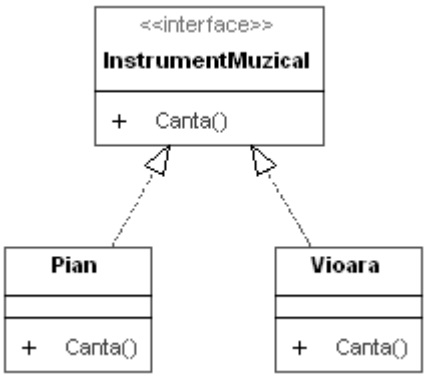


Figura 5.16. *Notaţia pentru interfeţe*

2.2.7. Trăsături statice

În notația UML, trăsăturile (atributele/câmpurile şi operațiile/metodele) statice se subliniază, ca în figura 5.17.

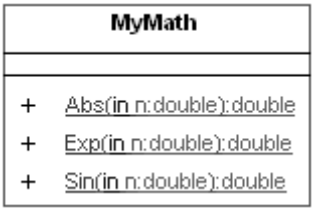


Figura 5.17. *Notaţia pentru trăsături statice*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

2.3. Diagrame de activităţi

Diagramele de activități sunt folosite pentru modelarea proceselor sau a algoritmilor din spatele unui anumit caz de utilizare. Notația este următoarea:

- *nod inițial:* un cerc plin; este punctul de start al diagramei;
- *nod final:* un cerc plin înconjurat de un alt cerc; o diagramă poate avea 0, 1 sau mai multe noduri finale;
- *acțiuni:* dreptunghiurile rotunjite reprezintă paşii activi executați în cadrul procesului;
- *arce:* săgețile diagramei;
- *punct final al fluxului:* un cerc cu un X în interior; indică faptul că procesul se opreşte în acest punct;
- *ramificație* (engl. "fork"): o bară neagră cu un flux de intrare şi mai multe fluxuri de ieşire; denotă începutul unor activități desfăşurate în paralel;
- *reunire* (engl. "join"): o bară neagră cu mai multe fluxuri de intrare şi un flux de ieşire; denotă sfârşitul prelucrărilor paralele;
- *condiție:* text asociat unui flux care defineşte o propoziție cu o valoare de adevăr şi care trebuie să fie adevărată pentru continuarea execuției;
- *decizie:* un romb cu un flux de intrare şi mai multe fluxuri de ieşire; fluxurile de ieşire includ condiții;
- *îmbinare* (engl. "merge"): un romb cu mai multe fluxuri de intrare şi un flux de ieşire; toate fluxurile de intrare trebuie să atingă acest punct pentru ca procesul să continue;
- *partiție* sau *culoar* (engl. "partition / swimlane"): o parte a diagramei care indică cine îndeplineşte acțiunile;
- *notă:* un comentariu care poate aduce informații suplimentare privind scopul, utilizarea sau constrângerile unei entități.

În figura 5.18 este prezentată o diagramă de activități cu decizii. „Candidatul trebuie să fie admis" este o notă asociată unei decizii.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

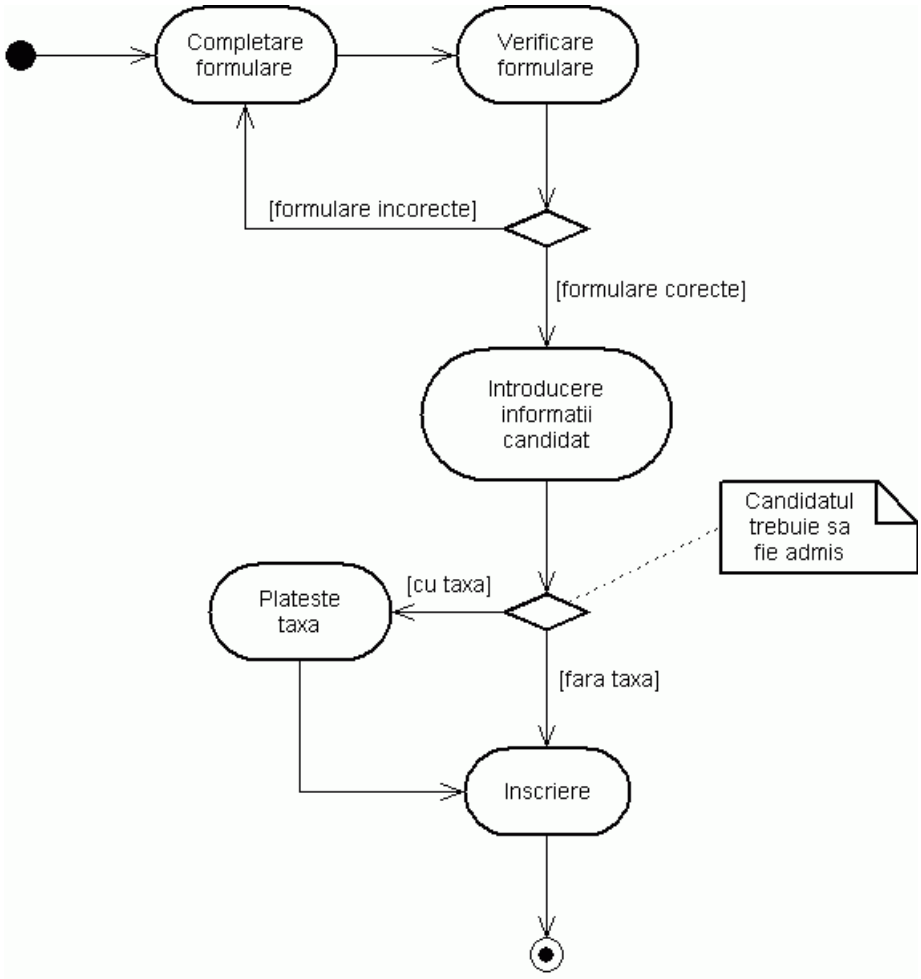


Figura 5.18. *Diagramă de activităţi cu decizii*

În figura 5.19 este prezentată o altă diagramă de activități, cu partiții şi ramificații.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

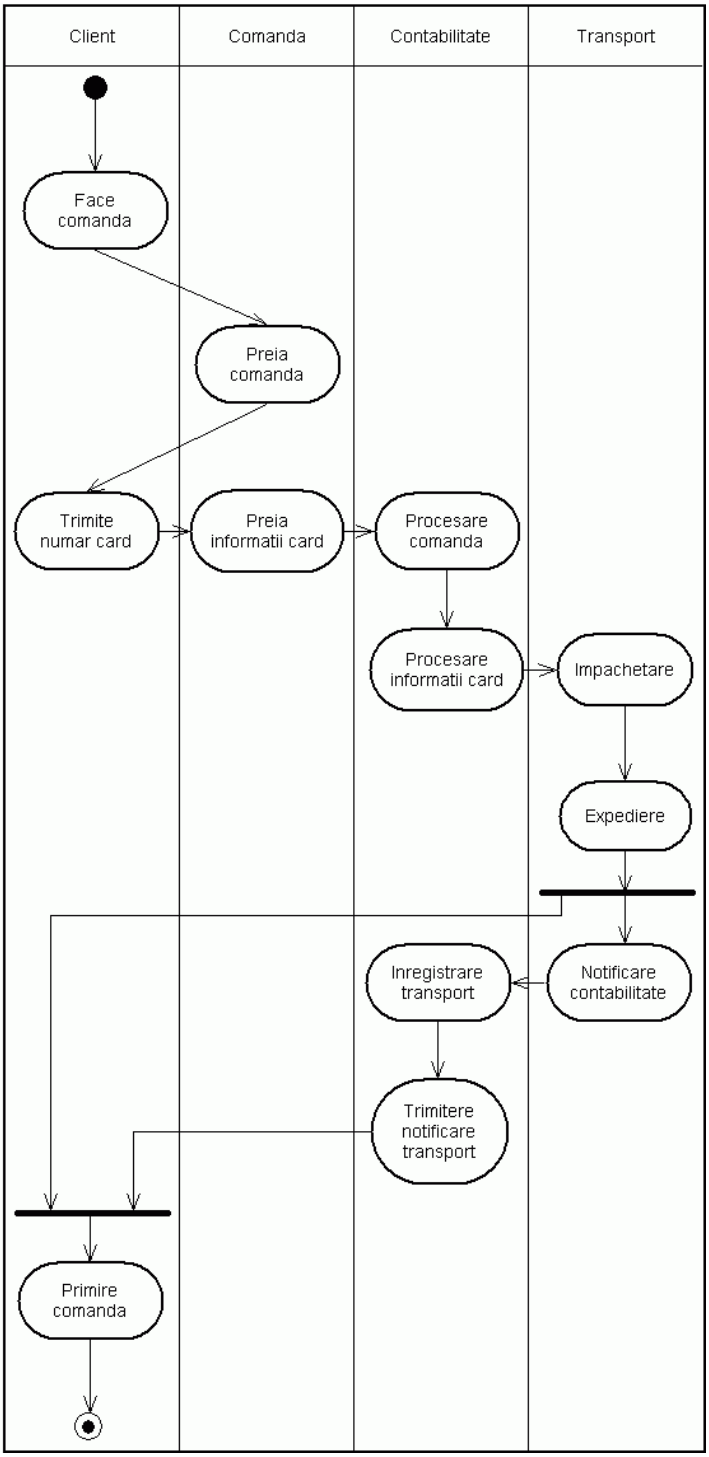


Figura 5.19. *Diagramă de activităţi cu partiţii şi ramificaţii*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

2.4. Diagrama de secvenţe

Diagrama de secvențe pune accentul pe aspectul temporal (ordonarea mesajelor). Notația grafică este un tabel care are pe axa *X* obiecte, iar pe axa *Y* mesaje ordonate crescător în timp. Axa *Y* arată pentru fiecare obiect timpul ca o linie verticală punctată, numită „linia vieții" unui obiect (engl. "lifeline") şi perioada în care obiectul deține controlul execuției (reprezentată printr-un dreptunghi) şi efectuează o acțiune, direct sau prin intermediul procedurilor subordonate.

În figura 5.20 este descrisă interacțiunea dintre doi abonați ai unei rețele de telefonie. De remarcat că în diagrama de secvențe utilizăm obiecte, nu clase. Într-un program pot exista mai multe instanțe ale aceleiaşi clase care au roluri diferite în sistem. Un obiect este identificat de numele său şi numele clasei pe care o instanțiază. Numele obiectului poate să lipsească dacă nu este semnificativ pentru înțelegerea comportamentului sistemului. Liniile orizontale continue semnifică mesaje inițiate de obiecte, iar liniile orizontale punctate reprezintă mesaje-răspuns.

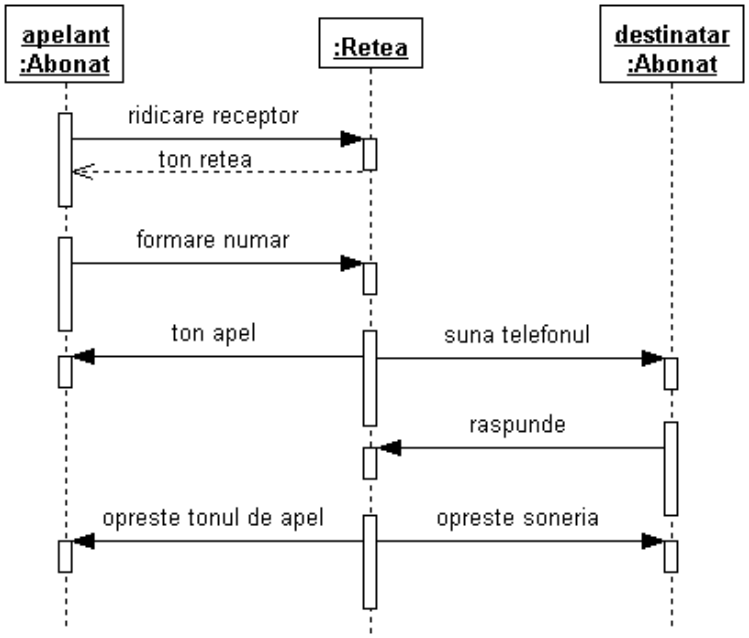


Figura 5.20. *Diagramă de secvențe*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

3. *Altova UModel*

Altova UModel este un instrument vizual pentru crearea de diagrame UML. Poate genera cod Java, C# şi Visual Basic .NET pe baza diagramelor şi poate să realizeze diagrame UML ale programelor existente. Este posibilă de asemenea ajustarea codului existent prin modificarea diagramelor corespunzătoare.

Fiecare tip de diagramă are o bară de instrumente corespunzătoare cu elementele UML caracteristice, care pot fi introduse în fereastra de desenare, conectate şi modificate (figura 5.21).

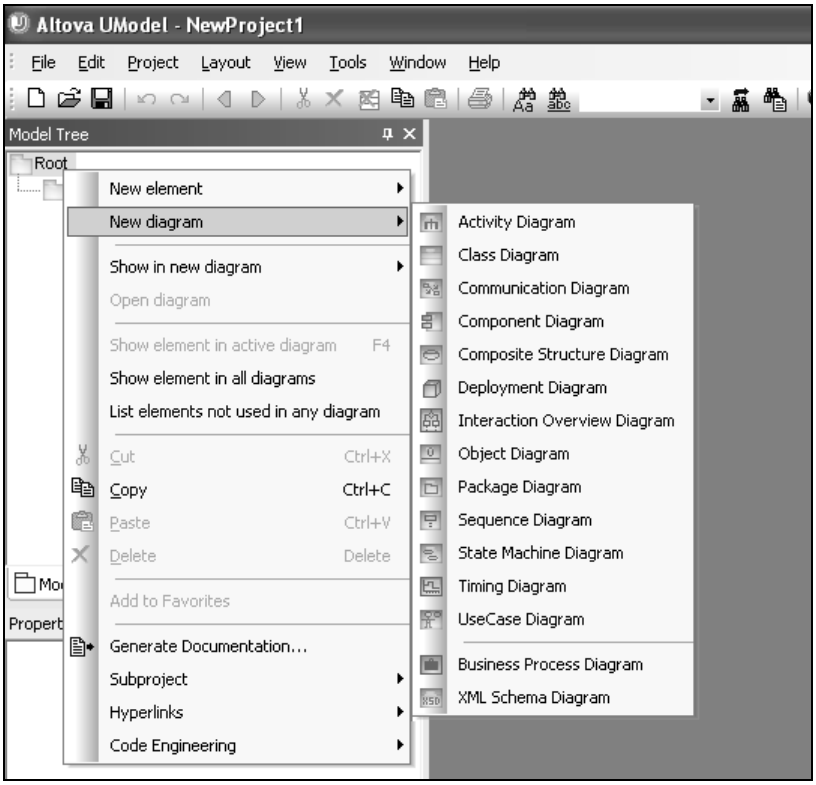


Figura 5.21. *Diagramele UML în Altova UModel*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

3.1. Diagrama de clase

Proprietățile (câmpurile) şi operațiile (metodele) unei clase pot fi adăugate mai rapid cu ajutorul tastelor F7, respectiv F8. Din bara de instrumente sau din conectorii dreptunghiului asociat clasei se creează tipurile de legături dintre clase, de exemplu asociere, moştenire etc. Membrii unei clase pot fi redenumiți direct pe diagramă sau din fereastra *Model Tree*.

Implicit, diagramele au vizibilitatea marcată grafic iar clasele au prima căsuță colorată cu un gradient. Aspectul vizual al diagramelor poate fi configurat folosind stilurile: *View → Styles*. De exemplu, diagrama din figura 5.22 sus este echivalentă cu cea de jos, eliminând gradientul şi marcând proprietatea *Show Visibility* drept *UML Style* în loc de *UModel Style*. Alte proprietăți utile sunt: *Show Namespace*, *Show Stereotypes*.

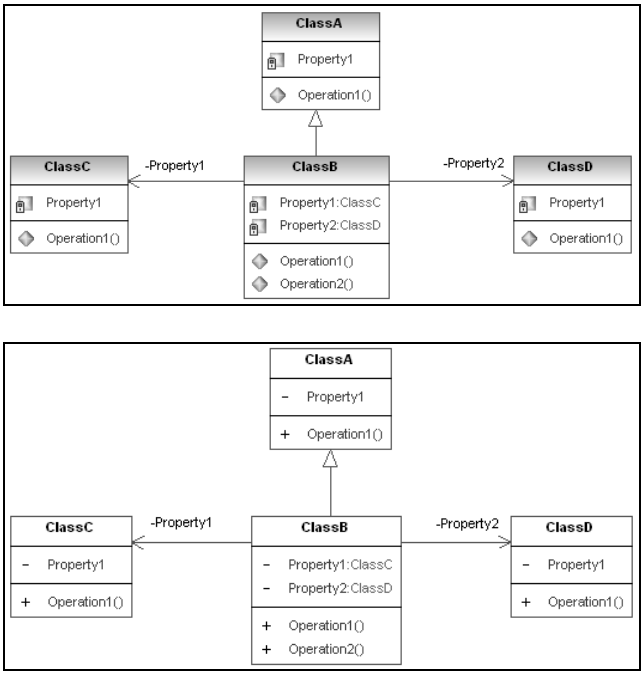


Figura 5.22. *Diagrame de clase cu diferite stiluri*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

3.1.1. Generarea de cod

Pentru a genera cod (C#) pe baza unei diagrame de clase, trebuie mai întâi creat un pachet care va conține clasele. În *Model Tree*, click dreapta pe *Root*, *New Element → Package*. Pe acest pachet trebuie aplicat profilul limbajului dorit pentru generare: click dreapta pe pachetul creat, *Code Engineering → Set as C# Namespace Root*.

În pachet se introduce apoi o diagramă de clase şi se adaugă clasele.

Tipurile proprietăților se pot completa din lista de tipuri din modelul ales, la apăsarea „ : ” după numele proprietății (figura 5.23).

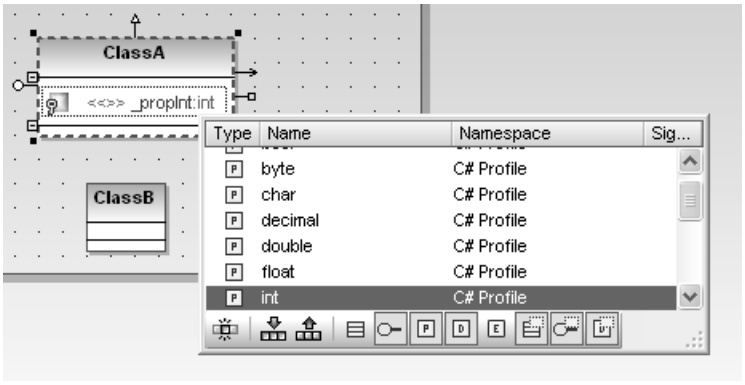


Figura 5.23. *Tipurile proprietăților*

Analog, pentru operații se vor adăuga parametrii şi tipul de return (figura 5.24). Numele parametrilor sunt precedate de cuvintele cheie in (parametru normal), out (parametru out), inout (parametru ref).

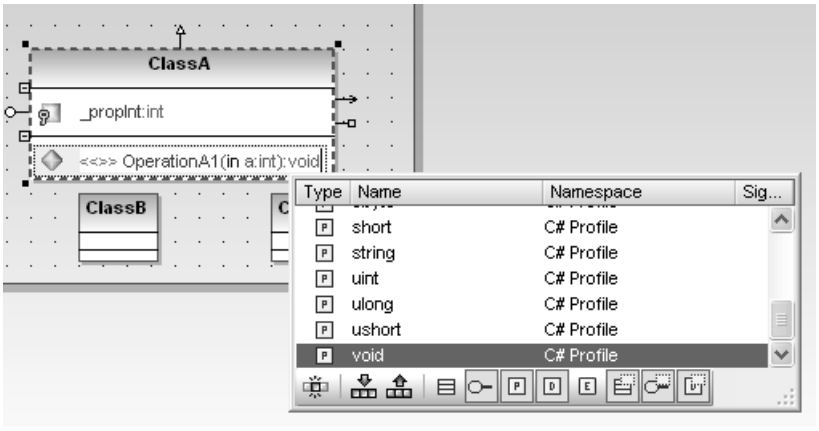


Figura 5.24. *Parametrii şi tipul de return pentru operații*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Pentru o clasă se pot adăuga automat accesori prin click dreapta, *Create getter/setter Operations...* (figura 5.25).

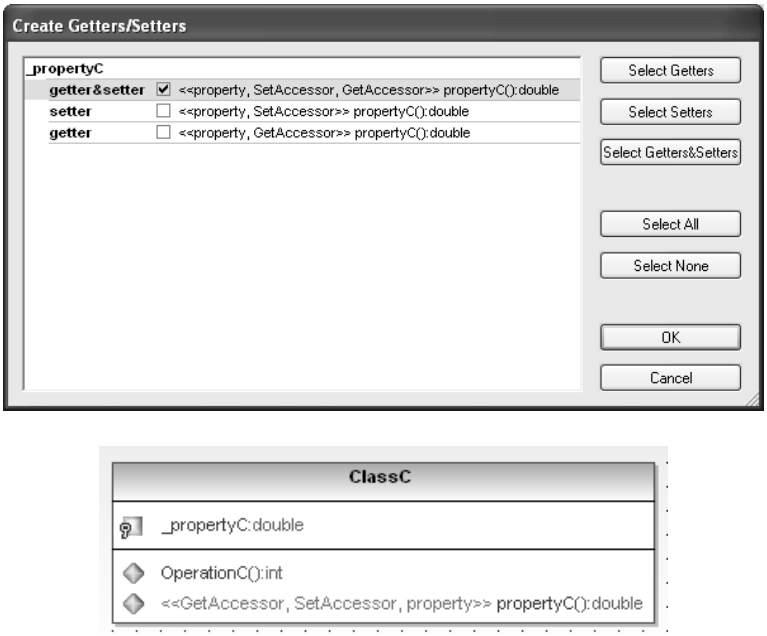


Figura 5.25. *Adăugarea de accesori*

Caracteristicile specifice ale unei operații se pot introduce din fereastra *Properties*, de exemplu cu stereotipurile <<constructor>>, <<virtual>>, <<abstract>>, <<override>> etc. (figura 5.26).

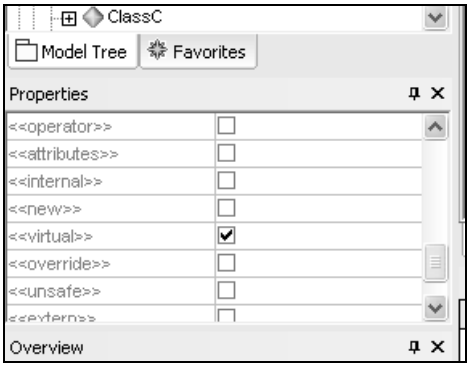


Figura 5.26. *Adăugarea de stereotipuri*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Trebuie precizat faptul că aceste stereotipuri devin disponibile doar după crearea unui pachet setat ca *C# Namespace Root*, în care au fost adăugate clasele.

Vizibilitatea membrilor se poate seta de asemenea apăsând pe imaginea din stânga numelui sau din fereastra *Properties*.

Se finalizează diagrama de clase (figura 5.27), care se poate exporta şi ca imagine din *File → Save Diagram As Image...*

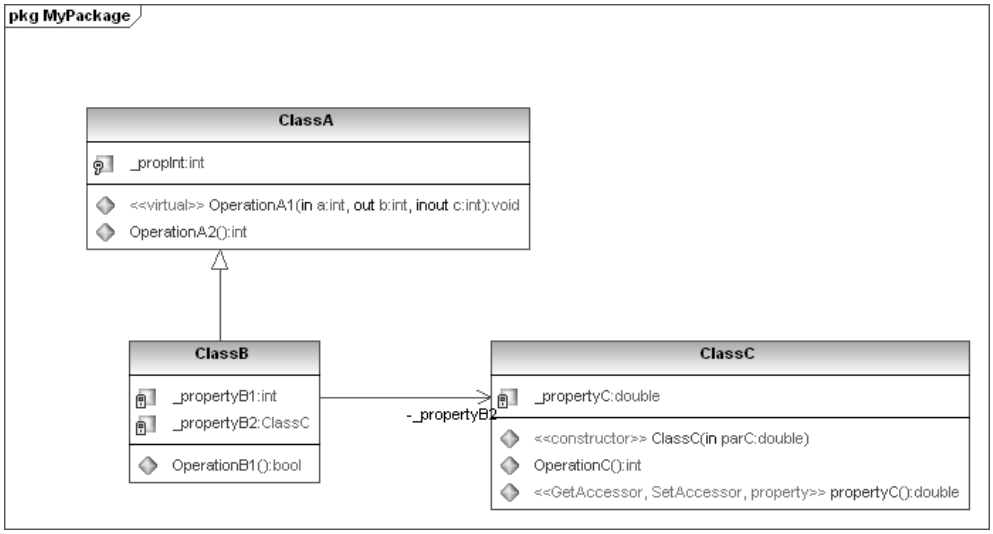


Figura 5.27. *Diagramă de clase cu asociere şi moştenire*

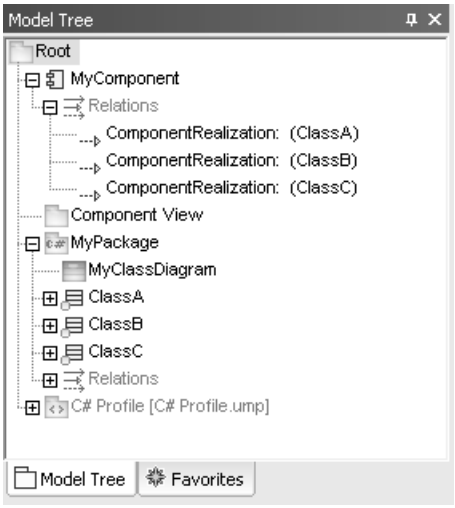


Figura 5.28. *Adăugarea unui component*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Pentru generarea efectivă a codului, mai trebuie adăugat un component în proiect, deoarece numai un astfel de element conține în viziunea UML codul propriu-zis pentru implementare: click dreapta pe *Root*, *New Element → Component*. După crearea componentului, se trag (*drag and drop*) clasele din diagrama de clase în component. Automat se creează niște relații de realizare (figura 5.28).

Pentru component mai trebuie specificate în fereastra *Properties* limbajul de programare dorit şi calea către directorul unde se vor genera fişierele sursă (figura 5.29).

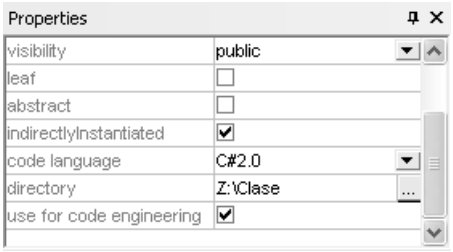


Figura 5.29. *Proprietățile componentului*

Apoi se face click dreapta pe numele componentului (aici *MyComponent*), *Code Engineering → Override Program Code from UModel Component...* (figura 5.30).

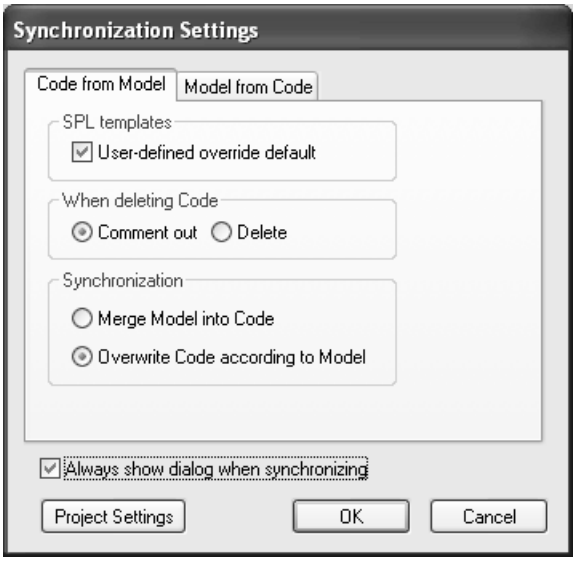


Figura 5.30. *Setările pentru sincronizare*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Scheletul de program generat pentru exemplul de mai sus este următorul:

```
public class ClassA
{
   protected int _propInt;

   public virtual void OperationA1(int a, out int b, ref int c)
   {
   }

   public int OperationA2()
   {
   }
}

public class ClassB : ClassA
{
   private int _propertyB1;
   private ClassC _propertyB2;

   public bool OperationB1()
   {
   }
}

public class ClassC
{
   private double _propertyC;

   public int OperationC()
   {
   }

   public double PropertyC
   {
      set { }
      get { }
   }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

3.1.2. Crearea diagramei unui proiect existent

Importarea de cod pentru crearea diagramei de clase este destul de simplă (figura 5.31); se alege *Project → Import Source Directory...* pentru a prelua fişierele sursă din directorul specificat (posibil recursiv) sau *Project → Import Source Project...* pentru importarea unei soluții Visual Studio (fişierul *sln*).

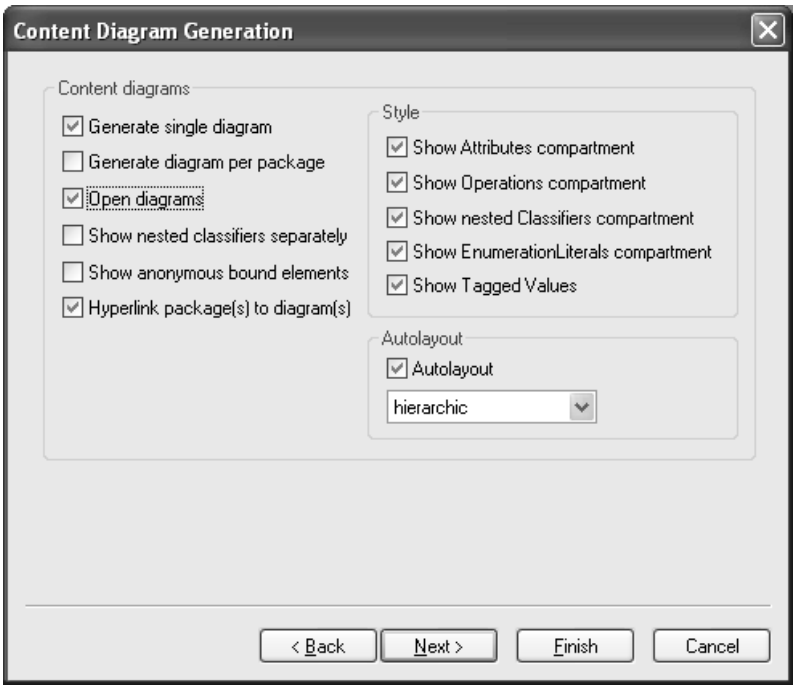


Figura 5.31. *Setările pentru generarea unei diagrame din cod*

Dacă se importă codul generat anterior, se observă că UModel afişează automat relațiile de generalizare (moştenire) dar nu şi pe cele de asociere (sau agregare/compunere). Acestea pot fi indicate în diagramă prin click dreapta pe un câmp şi alegerea opțiunii de afişare ca asociere (figura 5.32).

Relațiile de dependență nu apar automat, însă într-o diagramă nu trebuie să existe clase izolate. Ar însemna că acestea nu sunt utilizate deloc şi atunci nu se justifică prezența lor.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

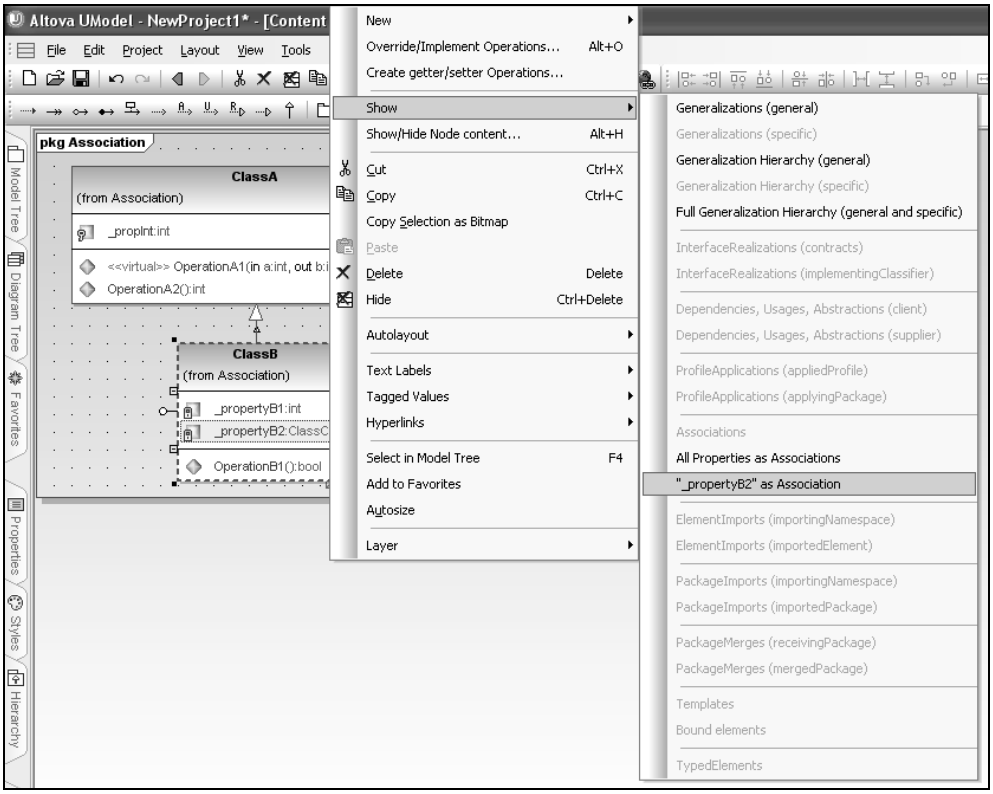


Figura 5.32. *Afişarea unei proprietăţi ca asociere*

Să considerăm următorul program:

```
namespace Dependency
{
   public class A
   {
      public static int Add(int a, int b)
      {
         return a + b;
      }
   }

   public class B
   {
      private int _x, _y;
      private int _sum;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    public int Sum
    {
      get { return _sum; }
    }

    public B(int x, int y)
    {
      _x = x; _y = y;
      _sum = A.Add(_x, _y);
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      B b = new B(1, 2);
      Console.WriteLine(b.Sum);
    }
  }
}
```

La importarea sa, diagrama UModel este cea din figura 5.33.

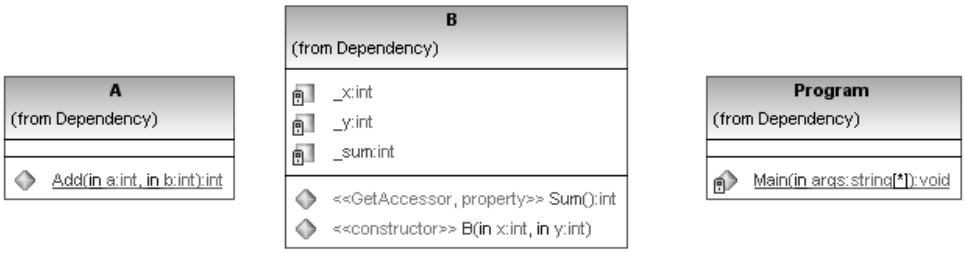


Figura 5.33. *Clasele importate*

În acest caz, relațiile de dependență între Program şi B, respectiv între B şi A trebuie trasate manual (figura 5.34).

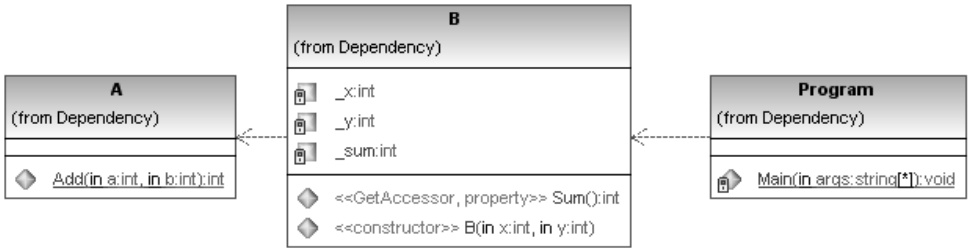


Figura 5.34. *Adăugarea relațiilor de dependență*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

3.1.3. Aranjarea automată a elementelor din diagrame

O funcționalitate foarte utilă este dispunerea automată a elementelor în fereastră (click dreapta în fereastră, *Autolayout All* → *Force Directed* sau *Hierarchical*).

3.2. Celelalte diagrame

Prin click dreapta pe *Root*, *New Diagram* pot fi introduse în proiect orice diagrame UML, iar în bara de meniuri apar elementele UML specifice tipului respectiv de diagramă.

4. Aplicaţii

4.1. Generați fişiere de cod C# dintr-o diagramă de clase (se poate urmări exemplul prezentat mai sus).

4.2. Realizați diagrama de clase a unui proiect C# prin importare.

4.3. Desenați diagramele din figurile: 5.3, 5.6, 5.8, 5.10, 5.11, 5.15, 5.16, 5.17, 5.18, 5.19 şi 5.20.

Indicație: Notațiile pot fi particularizate prin modificarea proprietăților elementelor, de exemplu modificarea tipului implicit de asociere (figura 5.35).

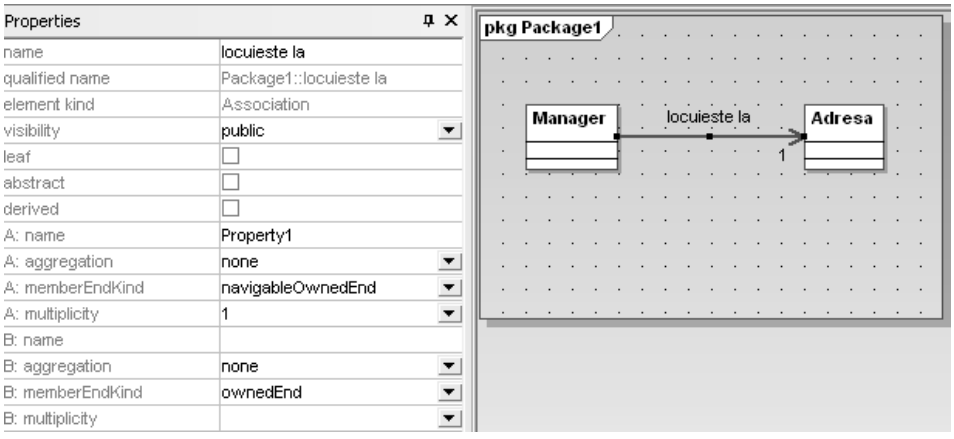


Figura 5.35. *Modificarea tipului de asociere*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

4.4. *Temă pentru acasă.* Desenați diagrama de activități şi diagrama de secvențe pentru un proiect C# la alegere.

Diagrama de secvențe trebuie realizată manual la un nivel mai înalt, nu diagrama realizată automat de versiunile recente ale UModel, în care reprezentarea este la nivel de linie de cod. Diagramele astfel rezultate sunt prea complexe pentru a fi înțelese.

Diagramele trebuie exportate şi în format imagine, de preferință *png*.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 6

Arhitectura MVC

1. Obiective
2. Introducere. Arhitectura cu trei straturi
3. Arhitectura *MVC*
4. Arhitectura *MVP*
5. Aplicații

1. Obiective

Obiectivul principal al acestui capitol este implementarea unui program după şablonul arhitectural *Model-Vizualizare-Prezentator* (engl. "Model-View-Presenter", MVP), o variantă a arhitecturii clasice *Model-Vizualizare-Controlor* (engl. "Model-View-Controller", MVC).

Ca obiective detaliate, vom avea:

1. Obiective de proiectare: particularizarea arhitecturii *MVP* pentru o aplicație cu interfață de tip consolă, subliniind faptul că arhitectura este independentă de tipul interfeței cu utilizatorul;
2. Obiective de programare: realizarea unui meniu consolă structurat pe niveluri;
3. Obiective diverse: calcularea distanței între două puncte de pe suprafața Pământului definite de coordonatele lor geografice.

2. Introducere. Arhitectura cu trei straturi

Una din recomandările de bază ale ingineriei programării este structurarea arhitecturii unei soluții pe niveluri, adică împărțirea sistemului în mai multe componente ordonate ierarhic, fiecare cu limitări legate de modul de interacțiune.

Din punct de vedere al terminologiei, se foloseşte „strat" (engl. "tier") pentru a indica o separare fizică a componentelor, adică *assembly*-uri (*dll*, *exe*) diferite pe aceeaşi maşină sau pe maşini diferite. Termenul de „nivel" (engl. "layer") indică o separare logică a componentelor, de exemplu *namespace*-uri diferite.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

O abordare simplă şi des întâlnită este utilizarea unei arhitecturi cu două straturi (engl. "two-tier architecture"). În acest caz, aplicația separă stratul de *prezentare* de stratul *datelor* aplicației. Datele reprezintă entitățile care definesc problema şi de obicei corespund unor tabele în baze de date. Clasele de prezentare au responsabilități precum recepționarea intrărilor de la utilizator (texte introduse, apăsarea unor butoane, alegerea unor opțiuni din meniuri etc.), apelurile către stratul de date, deciziile privind informațiile care vor fi arătate utilizatorului şi afişarea ieşirilor (texte, grafice etc.). Aceste responsabilități sunt destul de numeroase şi, pe măsură ce sistemul evoluează, stratul de prezentare poate deveni supraîncărcat.

O soluție naturală este divizarea acestui strat prea extins în două alte straturi: de *prezentare*, pentru preluarea intrărilor şi afişarea ieşirilor, şi respectiv de *logică a aplicației*, pentru asigurarea comunicațiilor cu stratul de acces la date şi pentru luarea deciziilor de control. Stratul de logică include toate prelucrările efective care manipulează datele interne şi pe cele ale utilizatorului, cu ajutorul algoritmilor specifici, pentru a controla fluxul aplicației.

Figura 6.1 prezintă comparativ arhitecturile cu două şi trei straturi.

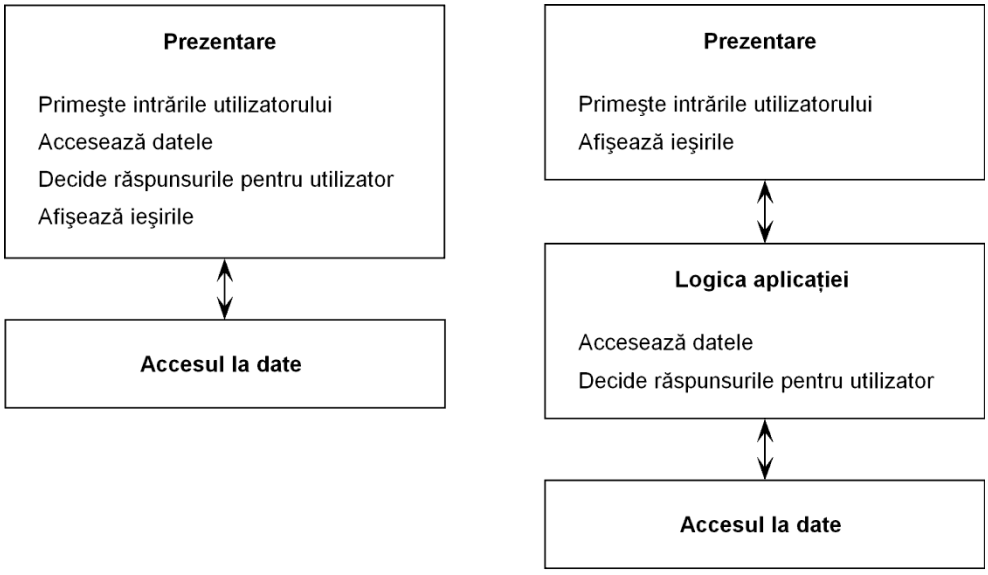


Figura 6.1. *Arhitectura cu două straturi (stânga). Arhitectura cu trei straturi (dreapta)*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

3. Arhitectura *MVC*

Arhitectura cu două straturi, în care se consideră că schimbul de informații are loc între interfața cu utilizatorul şi bazele de date (sau în general orice modalitate de stocare a datelor, inclusiv fişiere text, *xml* etc.) presupune ca din interfață să se acceseze şi să se modifice direct datele.

Totuşi, această abordare are câteva probleme semnificative. În primul rând, interfața cu utilizatorul se schimbă de obicei mai des decât baza de date. În al doilea rând, majoritatea aplicațiilor conțin cod funcțional (logica) ce realizează prelucrări mult mai complexe decât simpla transmitere de date.

Şablonul arhitectural *Model-Vizualizare-Controlor* izolează interfața, codul funcțional şi datele, astfel încât modificarea unuia din aceste trei componente să nu le afecteze pe celelalte două. O aplicație bazată pe şablonul *MVC* va avea trei module corespunzătoare:

1. **Model:** conține datele, starea şi logica aplicației. Deşi nu cunoaşte Controlorul şi Vizualizarea, furnizează o interfață pentru manipularea şi preluarea stării şi poate trimite notificări cu privire la schimbarea stării. De obicei primeşte cereri privind starea datelor de la Vizualizare şi instrucțiuni de modificare a datelor sau stării de la Controlor;
2. **Vizualizare:** afişează Modelul într-o formă potrivită pentru utilizator. Pentru un sigur Model pot exista mai multe Vizualizări, de exemplu o listă de elemente poate fi afişată într-un control vizual precum *ListBox*, într-o consolă sau într-o pagină web;
3. **Controlor:** primeşte intrările de la utilizator şi apelează obiectele Modelului pentru a prelucra noile informații.

Există mai multe variante ale arhitecturii *MVC*, însă în general fluxul de control este următorul:

1. Utilizatorul interacționează cu interfața aplicației, iar Controlorul preia intrarea şi o interpretează ca pe o acțiune ce poate fi recunoscută de către Model;
2. Controlorul trimite Modelului acțiunea utilizatorului, ceea ce poate conduce la schimbarea stării Modelului;
3. În vederea afişării rezultatului de către Vizualizare, Controlorul îi poate trimite acesteia o cerere de actualizare, sau Modelul îi trimite o notificare privind schimbarea stării sale;

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

4. Vizualizarea preia datele necesare din Model şi le afişează.

În continuare, aplicația aşteaptă o nouă acțiune a utilizatorului iar ciclul se reia.

Trebuie precizat că Modelul nu este doar o bază de date, ci încapsulează şi logica domeniului necesară pentru manipularea datelor din aplicație. De multe ori se foloseşte şi un mecanism de stocare persistentă a acestora, de exemplu într-o bază de date, însă arhitectura *MVC* nu menționează explicit stratul de acces la date; acesta se consideră implicit ca parte din Model.

Figura 6.2 prezintă relațiile structurale între cele trei module.

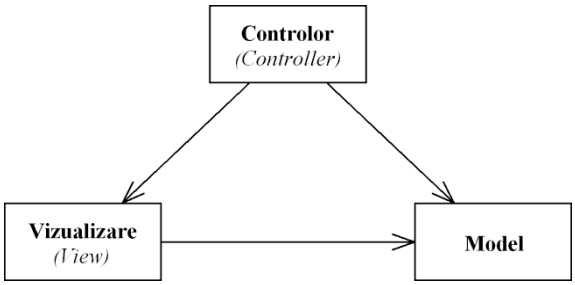


Figura 6.2. *Relaţiile structurale între componentele arhitecturii MVC*

Pentru o aplicație mai simplă, aceste module pot reprezenta clase. Din punctul de vedere al implementării, săgețile de asociere înseamnă că:

- Vizualizarea va avea un câmp de tip Model; de obicei va primi ca parametru în constructor o referință la obiectul Model;
- Controlorul va avea două câmpuri de tip Vizualizare şi Model; de obicei va primi ca parametri în constructor referințe la obiectele Vizualizare şi Model.

Mai ales în aplicațiile web, este clar definită separația dintre Vizualizare (*browser*-ul) şi Controlor (componentele server care răspund cererilor *http*).

Prin separarea celor trei funcționalități se atinge o cuplare slabă între module, caracteristică dorită în toate programele deoarece modificările dintr-o secțiune a codului nu necesită modificări şi în alte secțiuni. Decuplarea scade complexitatea proiectării şi creşte flexibilitatea şi potențialul de reutilizare.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Avantajele principale ale şablonului *MVC* sunt următoarele:

- **Modificări rapide.** Clasele şablonului trebuie doar să implementeze nişte interfeţe prestabilite, astfel încât acestea să cunoască metodele pe care le pot apela în celelalte clase. Când se doresc modificări, nu trebuie rescrisă o clasă, se poate implementa una nouă şi se poate utiliza direct, chiar alături de una veche. De asemenea, Vizualizările şi Modelele existente pot fi refolosite pentru alte aplicaţii cu un Controlor diferit.
- **Modele de date multiple.** Modelul nu depinde de nicio altă clasă din şablon. Datele pot fi stocate în orice format: text, *xml* sau baze de date Access, Oracle, SQL Server etc.;
- **Interfeţe multiple.** Deoarece Vizualizarea este separată de Model, pot exista în aplicaţie mai multe tipuri de Vizualizări ale aceloraşi date. Utilizatorii pot alege mai multe scheme de afişare: mai multe *skin*-uri sau comunicarea în mai multe limbi. Aplicaţia poate fi extinsă uşor pentru a include moduri de vizualizare complet diferite: consolă, interfaţă grafică cu utilizatorul în ferestre (*desktop*), documente web sau pentru PDA-uri.

4. Arhitectura *MVP*

În abordarea clasică, descrisă de Trygve Reenskaug pe când lucra la limbajul Smalltalk la Xerox PARC (1978-1979), logica este în Model, iar Controlorul gestionează intrările de la utilizator.

Pentru aplicaţiile de tip consolă, este relativ simplu ca intrările să fie preluate de Controlor iar afişarea să se facă de către Vizualizare, pe baza datelor din Model. Pentru aplicaţiile moderne, cu interfeţe grafice cu utilizatorul (GUI) precum ferestrele Windows, clasele de vizualizare sunt cele care primesc intrările utilizatorului. De aceea, în astfel de situaţii, Vizualizarea şi Controlorul nu mai sunt clar delimitate.

În *Cocoa* (unul din mediile de dezvoltare de aplicaţii orientate obiect native ale Apple pentru *Mac OS X*) şi *DDay.MVC* (proiectul *DDay* reprezintă o colecţie de biblioteci open-source pentru tehnologiile .NET), Controlorul conţine logica aplicaţiei.

Pentru a răspunde noilor realităţi ce privesc interacţiunea utilizatorilor cu interfeţele aplicaţiilor, a fost propus şablonul *Model-Vizualizare-Prezentator*, *MVP*. Aici, stratul de *prezentare* constă în obiecte

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

de Vizualizare iar *logica aplicației* constă în obiecte de control (Prezentator/Controlor). Pentru fiecare obiect de vizualizare există un obiect de control.

Din punct de vedere al terminologiei, se poate utiliza termenul de Controlor pentru Prezentator în şablonul *MVP*, deoarece acesta poate fi considerat o variantă modernă a şablonului clasic *MVC*.

Deşi ambele şabloane, *MVC* şi *MVP*, se bazează pe principiul comun al arhitecturii cu trei straturi, acestea au două diferențe majore:

1. În *MVC*, Controlorul primeşte şi prelucrează intrările de la utilizator iar în *MVP*, Vizualizarea primeşte intrările şi apoi deleagă prelucrările către Controlorul corespunzător;
2. În *MVC*, Vizualizarea primeşte notificări privind schimbările Modelului şi afişează noile informații pentru utilizator. În *MVP*, Controlorul modifică direct Vizualizarea, ceea ce face şablonul *MVP* mai uşor de folosit decât şablonul *MVC*.

Aceste diferențe fac şablonul *MVP* mai atractiv decât şablonul *MVC* pentru aplicațiile din prezent.

4.1. Variante de actualizare a *Vizualizării*

Când Modelul este actualizat, Vizualizarea trebuie de asemenea actualizată pentru a reflecta modificările. Actualizarea Vizualizării poate fi realizată în două variante: *Vizualizarea pasivă* (engl. "Passive View") şi *Controlorul supervizor* (engl. "Supervising Controller"). Figura 6.3 ilustrează modelele logice ale celor două variante.

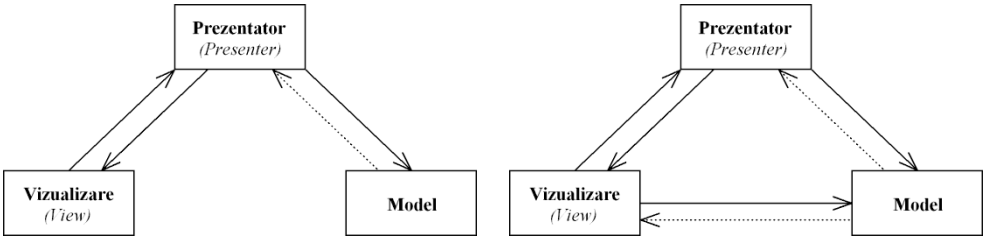


Figura 6.3. *Vizualizarea pasivă (stânga) şi Controlorul supervizor (dreapta)*

În abordarea *Vizualizării pasive*, Prezentatorul actualizează Vizualizarea pentru a reflecta schimbările din Model. Interacțiunea cu

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Modelul este gestionată exclusiv de Prezentator iar Vizualizarea nu îşi poate da seama direct de schimbările din Model. La rândul său, Vizualizarea este actualizată exclusiv de către Prezentator.

Abordarea este utilă atunci când Prezentatorul trebuie să realizeze unele prelucrări complexe asupra datelor, înainte de a afişa informaţiile pentru utilizator. Acestea apar de exemplu atunci când stările controalelor din interfaţa grafică depind de anumite prelucrări ale datelor. Să considerăm cazul în care un utilizator doreşte să împrumute o carte a unui anumit autor de la bibliotecă. Dacă toate cărţile autorului respectiv sunt deja împrumutate, butonul *Împrumută* poate fi dezactivat. În această situaţie, faptul că împrumutul este imposibil nu este reţinut în Model, ci este determinat de Prezentator, care apoi îi cere Vizualizării să dezactiveze butonul respectiv.

În abordarea *Controlorului supervizor*, Vizualizarea interacţionează direct cu Modelul pentru a transfera date, fără intervenţia Prezentatorului. Prezentatorul actualizează Modelul şi manipulează starea Vizualizării doar în cazurile în care există o logică complexă a interfeţei cu utilizatorul. Vizualizarea poate fi actualizată şi direct, pe baza modificărilor datelor din Model.

Astfel de transferuri simple, în care Vizualizarea comunică direct cu Modelul, sunt în general situaţiile în care se schimbă unele date în Model iar acestea sunt preluate şi afişate în Vizualizare, fără prelucrări suplimentare. De exemplu, interfaţa afişează o listă de cărţi împrumutate de un student de la bibliotecă, listă păstrată de Model într-o bază de date. Când studentul împrumută o nouă carte, baza de date se schimbă iar Vizualizarea preia direct din Model elementele listei pentru afişare.

Vizualizarea pasivă este asemănătoare arhitecturii cu trei straturi tipice, în care stratul de logică a aplicaţiei se interpune între stratul de prezentare şi stratul de acces la date.

Decizia asupra alegerii uneia din cele două variante depinde de priorităţile aplicaţiei. Dacă este mai importantă testabilitatea, *Vizualizarea pasivă* este mai potrivită, deoarece se poate testa toată logica interfeţei cu utilizatorul prin testarea Prezentatorului. Pe de altă parte, dacă simplitatea este mai importantă, *Controlorul supervizor* este o opţiune mai bună deoarece, pentru schimbări mici în interfaţă, nu mai trebuie inclus cod în Prezentator pentru actualizarea Vizualizării. Astfel, *Controlorul supervizor* necesită de obicei mai puţin cod întrucât Prezentatorul nu mai efectuează actualizările simple ale Vizualizării.

Şabloanele *MVC* şi *MVP* au scopuri similare, însă diferă prin modalităţile în care îşi ating aceste scopuri.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

5. Aplicații

5.1. Realizați un program de tip consolă cu arhitectura *MVP, Controlor supervizor* pentru determinarea costurilor unei firme de transport. Se vor putea calcula costurile de transport între două orașe, identificate prin nume, latitudine şi longitudine.

Aplicația va permite două roluri: administrator şi utilizator, cu funcții diferite (figura 6.4).

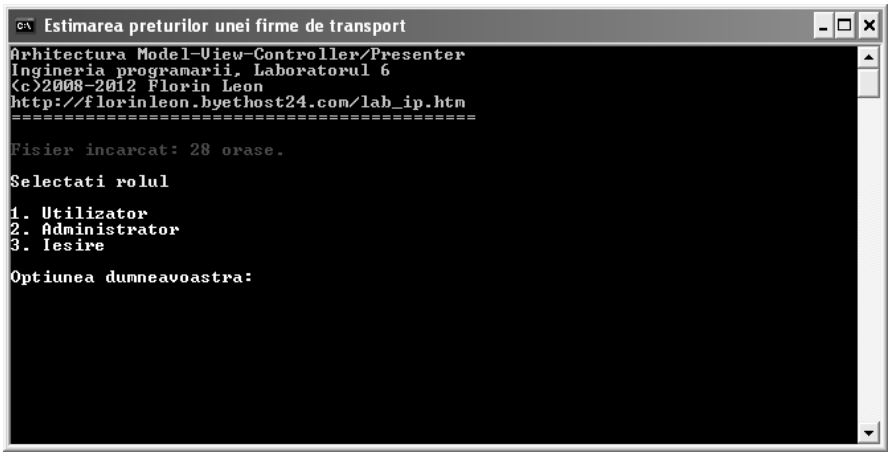


Figura 6.4. *Meniul principal*

Pentru rolul de utilizator comenzile disponibile sunt prezentate în figura 6.5.

Figura 6.5. *Meniul rolului de utilizator*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Pentru rolul de administrator comenzile disponibile sunt prezentate în figura 6.6.

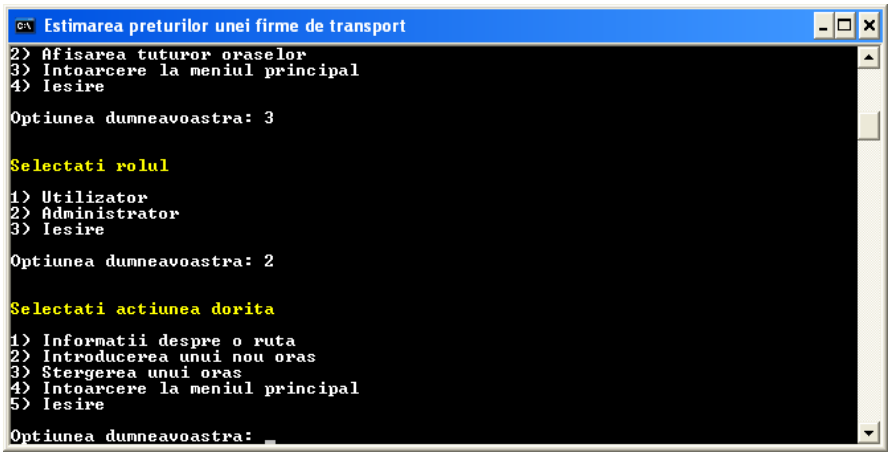


Figura 6.6. *Meniul rolului de administrator*

Indicații:

- Se furnizează codul sursă pentru Model, împreună cu un fișier text care conține mai multe orașe;

Model.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

namespace TransportInfo
{
   public class Model : IModel
   {
      #region Fields

      private const string CityFileName = "cities.txt";
      private List<City> _cityList;
      private bool _wasModified; // lista cu orașe va fi salvată în final doar dacă s-a modificat

      #endregion
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
#region Properties
public int CityCount
{
   get { return _cityList.Count; }
}
#endregion

#region Constructor

public Model()
{
   _cityList = new List<City>();
   _wasModified = false;
}

#endregion


#region Public Methods

public int GetNumberOfCities()
{
   return _cityList.Count;
}

public bool DataExists()
{
   if (!File.Exists(CityFileName))
   {
      _wasModified = true;
      return false;
   }
   else
      return true;
}

public void InitializeData()
{
   StreamReader sr = new StreamReader(CityFileName);
   string line;
   while ((line = sr.ReadLine()) != null)
      _cityList.Add(ParseCityLine(line));
   sr.Close();
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
public bool Add(City city)
{
   // dacă un oraş cu acelaşi nume există deja, el va fi şters
   bool overwrite = false;

   for (int i = 0; i < _cityList.Count; i++)
   {
      if (_cityList[i].Name.Trim().ToUpper() == city.Name.Trim().ToUpper())
      {
         _cityList.RemoveAt(i--);
         overwrite = true;
      }
   }

   // adăugarea noului oraş
   _cityList.Add(city);
   _wasModified = true;
   return !overwrite;
}

public bool Delete(string cityName)
{
   for (int i = 0; i < _cityList.Count; i++)
   {
      if (_cityList[i].Name == cityName)
      {
         _cityList.RemoveAt(i);
         _wasModified = true;
         return true;
      }
   }

   return false;
}

public bool Exists(string cityName)
{
   // dacă un oraş există
   for (int i = 0; i < _cityList.Count; i++)
   {
      if (_cityList[i].Name == cityName)
         return true;
   }

   return false;
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
public City Search(string cityName)
{
   // caută un oraş după nume şi returnează obiectul corespunzător
   for (int i = 0; i < _cityList.Count; i++)
   {
      if (_cityList[i].Name == cityName)
         return _cityList[i];
   }

   return new City();
}

public string ListAll()
{
   // creează un string cu numele tuturor oraşelor
   if (_cityList.Count == 0)
      return string.Empty;

   StringBuilder sb = new StringBuilder();
   sb.Append(_cityList[0].Name);

   for (int i = 1; i < _cityList.Count; i++)
   {
      sb.Append(", ");
      sb.Append(_cityList[i].Name);
   }

   return sb.ToString();
}

/// <summary>
/// Salvează datele doar dacă lista de oraşe s-a modificat
/// </summary>
/// <returns>Returnează true dacă noile date au fost salvate </returns>
public bool SaveData()
{
   // dacă datele s-au modificat, ele sunt salvate
   if (_wasModified)
   {
      StreamWriter sw = new StreamWriter(CityFileName);

      for (int i = 0; i < _cityList.Count; i++)
      {
         City c = _cityList[i];
         sw.WriteLine(c.Name + "\t" + c.Latitude + "\t" + c.Longitude);
      }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
            sw.Close();
            return true;
        }
        else
            return false;
    }

    #endregion

    #region Private Methods

    private static City ParseCityLine(string line)
    {
        // citeşte informaţiile unui oraş de pe o linie din fişier
        string[] toks = line.Split('\t');
        City city = new City(toks[0], Convert.ToDouble(toks[1]), Convert.ToDouble(toks[2]));
        return city;
    }

    #endregion
  }
}
```

cities.txt

```
Iasi            47.167  27.583
Bacau           46.567  26.917
Piatra Neamt    46.933  26.383
Suceava         47.667  26.183
Botosani        47.733  26.667
Vaslui          46.633  27.733
Bucuresti       44.44   26.1
Cluj-Napoca     46.78   23.59
Timisoara       45.76   21.23
Constanta       44.18   28.63
Brasov          45.66   25.61
Chisinau        47.033  28.833
Balti           47.75   27.917
Amsterdam       52.383  4.9
Atena           37.967  23.767
Belgrad         44.817  20.45
Berlin          52.533  13.4
Bruxelles       50.85   4.35
Budapesta       47.483  19.083
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Londra	51.517	-0.1
Madrid	40.417	-3.75
Moscova	55.75	37.583
Oslo	59.917	10.75
Praga	50.083	14.367
Paris	48.833	2.333
Roma	41.9	12.5
Sofia	42.75	23.333
Viena	48.2	16.367

- Se furnizează codul sursă pentru structura corespunzătoare unui oraş, împreună cu funcția de calcul al distanței;

City.cs

```
namespace TransportInfo
{
   public struct City
   {
      // readonly pentru ca structura să fie immutable
      // alternativa este abordarea cu câmpuri private şi proprietăţi publice
      public readonly double Latitude, Longitude;
      public readonly string Name;

      public City(string name, double latitude, double longitude)
      {
         Name = name;
         Latitude = latitude;
         Longitude = longitude;
      }
   }
}
```

Calculator.cs

```
namespace TransportInfo
{
   public class BusinessCalculator
   {
      #region Public Static Methods
      public static double Distance(City c1, City c2)
      {
         // calculează distanţa în kilometri între două puncte de pe suprafaţa Pământului
         // identificate prin latitudine şi longitudine utilizând coordonate sferice

         double a1 = c1.Latitude * Math.PI / 180.0;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
        double b1 = c1.Longitude * Math.PI / 180.0;
        double a2 = c2.Latitude * Math.PI / 180.0;
        double b2 = c2.Longitude * Math.PI / 180.0;
        const double EarthRadius = 6378; // raza Pământului în km

        return (int)(EarthRadius * Math.Acos(Math.Cos(a1) * Math.Cos(b1) * Math.Cos(a2)
            * Math.Cos(b2) + Math.Cos(a1) * Math.Sin(b1) * Math.Cos(a2) * Math.Sin(b2) +
            Math.Sin(a1) * Math.Sin(a2)));
      }

      public static double Cost(double distance)
      {
        // aici se poate introduce orice funcţie de calcul al costului
        double euro = 5 + distance / 30.0;
        return euro * 4.3;
      }
      #endregion
    }
}
```

- Întrucât cele trei meniuri au structuri similare, se recomandă crearea unei metode comune care să primească drept parametru lista de opțiuni posibile;
- Se recomandă ca opțiunile alese de utilizator să fie tratate ca o enumerație.

```
public enum UserChoice { AdminMenu, UserMenu, PreviousMenu, Route, AddCity,
    RemoveCity, Exit, List, Undefined };
public enum MenuState { Main, Administrator, User };
```

Metoda Main din clasa Program poate avea conținutul următor:

```
static class Program
{
    static void Main()
    {
        Model model = new Model();
        ConsoleView view = new ConsoleView(model);
        Presenter presenter = new Presenter(view, model);
        view.SetPresenter(presenter);
        view.Start();
    }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

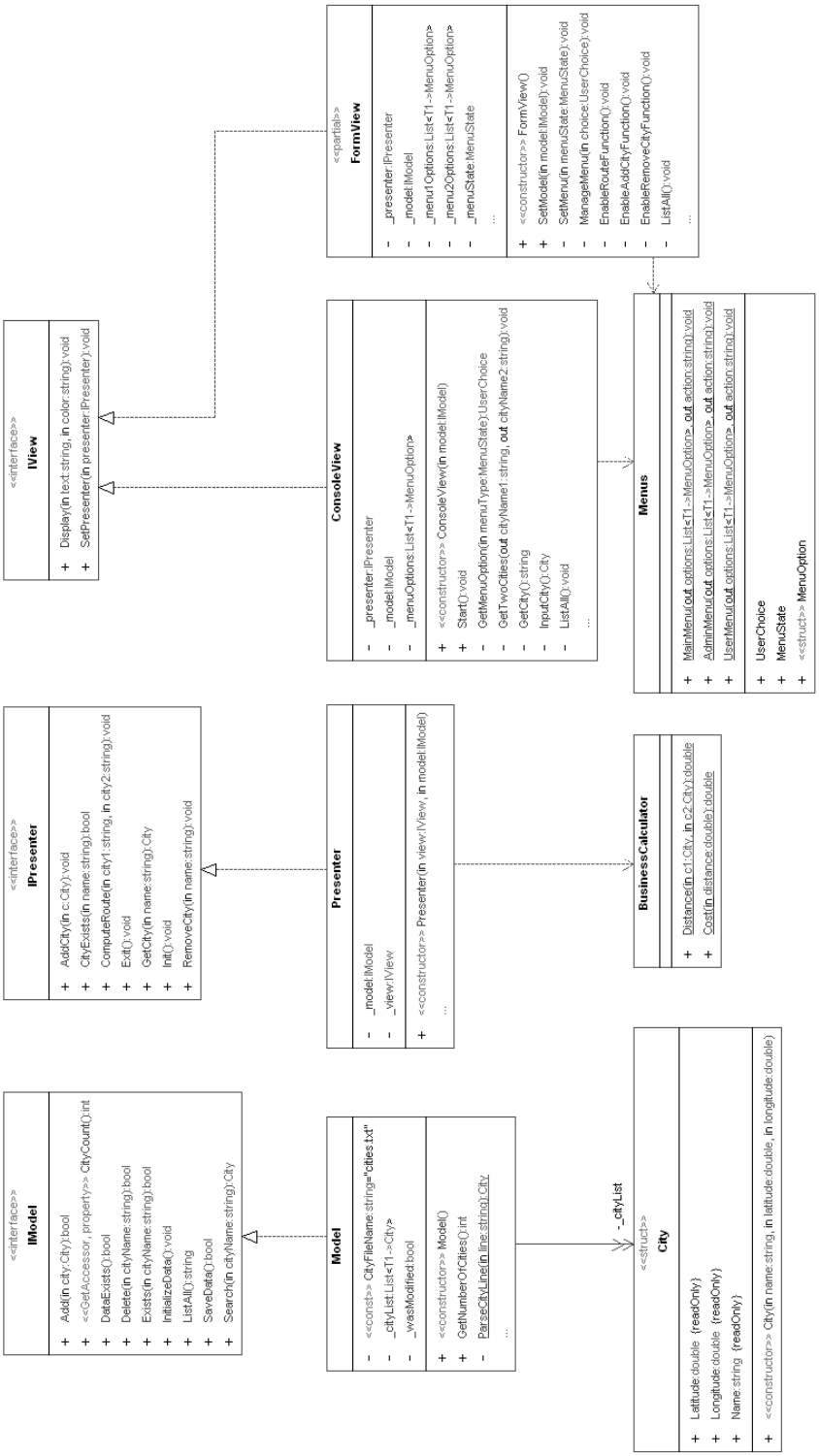


***Figura 6.7.** Exemplu de rezolvare: diagrama de clase*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Un exemplu de diagramă de clase pentru aplicație este prezentată în figura 6.7. Pentru creşterea clarității, nu s-au mai reprezentat explicit relațiile de asociere pentru câmpurile de tip IPresenter, IModel, IView şi nici numele metodelor din interfețe implementate de clasele concrete.

Metoda ListAll din clasele View apelează direct metoda ListAll din clasa Model. Aici apare diferența între abordarea *Controlorului supervizor* şi cea a *Vizualizării pasive*. Dacă s-ar fi utilizat cea de a doua abordare, ListAll din View ar fi apelat o metodă corespunzătoare din Presenter, iar aceasta ar fi apelat metoda ListAll din Model.

Proiectele soluției şi fişierele sursă pot fi structurate ca în figura 6.8.

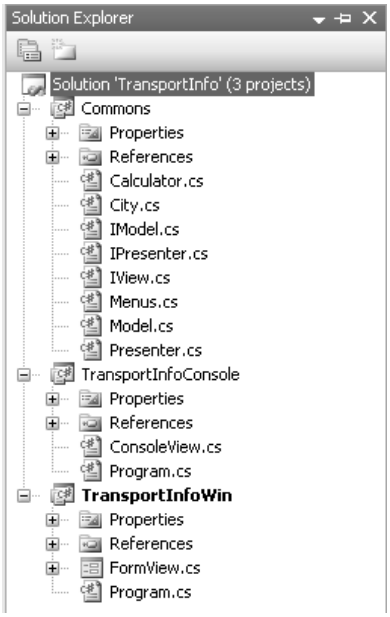


Figura 6.8. *Exemplu de rezolvare: structurarea soluției*

5.2. *Temă pentru acasă*. Păstrând aceleaşi clase pentru Model şi Prezentator, realizați o Vizualizare nouă, cu o interfață grafică de tip *Windows Forms*, cu aceeaşi funcționalitate ca şi aplicația consolă dezvoltată anterior.

Indicații: Metoda Main din clasa Program poate avea conținutul următor:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
static class Program
{
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);

        Model model = new Model();
        FormView view = new FormView();
        Presenter presenter = new Presenter(view, model);
        view.SetModel(model);
        view.SetPresenter(presenter);
        Application.Run(view);
    }
}
```

Pentru a forța utilizatorul să folosească doar opțiunea de *Ieşire* pentru a închide programul, butonul de închidere a ferestrei poate fi dezactivat cu ajutorul secvenței următoare de cod:

```
#region Disable Close X Button
const int MF_BYPOSITION = 0x400;

[DllImport("User32")]
private static extern int RemoveMenu(IntPtr hMenu, int nPosition, int wFlags);

[DllImport("User32")]
private static extern IntPtr GetSystemMenu(IntPtr hWnd, bool bRevert);

[DllImport("User32")]
private static extern int GetMenuItemCount(IntPtr hWnd);

private void FormView_Load(object sender, EventArgs e)
{
    IntPtr hMenu = GetSystemMenu(this.Handle, false);
    int menuItemCount = GetMenuItemCount(hMenu);
    RemoveMenu(hMenu, menuItemCount - 1, MF_BYPOSITION);
}
#endregion
```

Un exemplu de interfață grafică este prezentat în capturile ecran din figurile 6.9, 6.10 şi 6.11.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

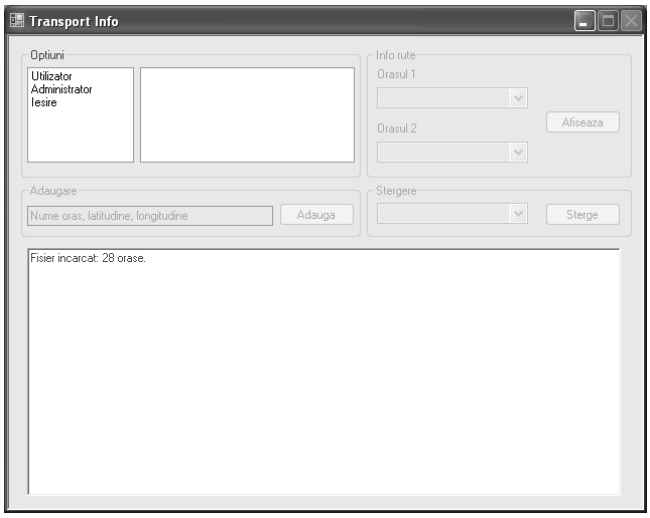


Figura 6.9. *Exemplu de rezolvare: meniul principal*

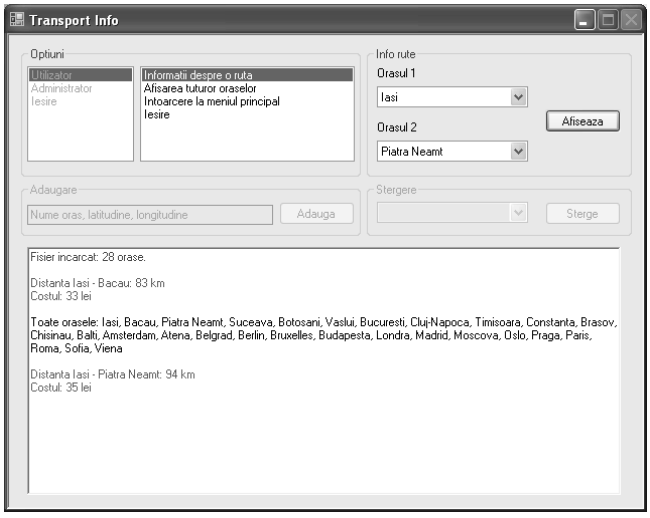


Figura 6.10. *Exemplu de rezolvare: rolul de utilizator*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

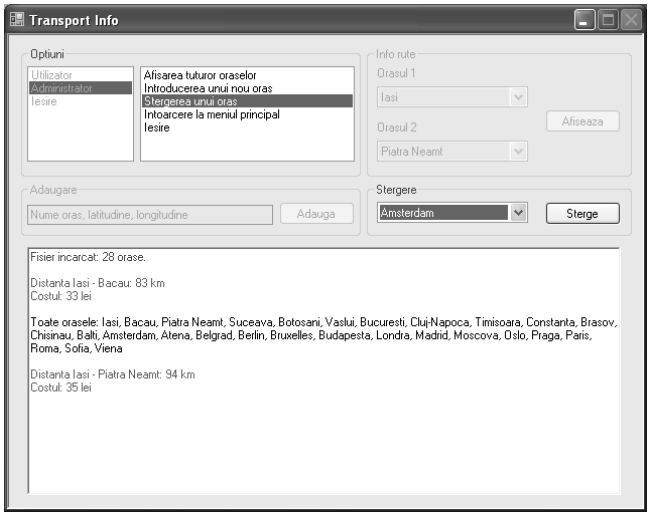


Figura 6.11. *Exemplu de rezolvare: rolul de administrator*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 7

Şablonul de proiectare Metoda Fabrică

1. Obiective
2. Şablonul creaţional *Metoda Fabrică*
3. Exemplu de implementare
4. Moştenirea şi polimorfismul
5. Aplicaţii

1. Obiective

Obiectivele capitolului 7 sunt următoarele:

- Implementarea unui program după şablonul de proiectare *Metoda Fabrică* (engl. "Factory Method");
- Precizarea unor noţiuni privind moştenirea şi polimorfismul (clase abstracte, interfeţe, membri virtuali), utilizate în majoritatea şabloanelor de proiectare.

2. Şablonul creaţional *Metoda Fabrică*

Şablonul *Metoda Fabrică* defineşte o interfaţă pentru crearea unui obiect, dar lasă subclasele să decidă ce clasă să instanţieze.

Diagrama de clase este prezentată în figura 7.1.

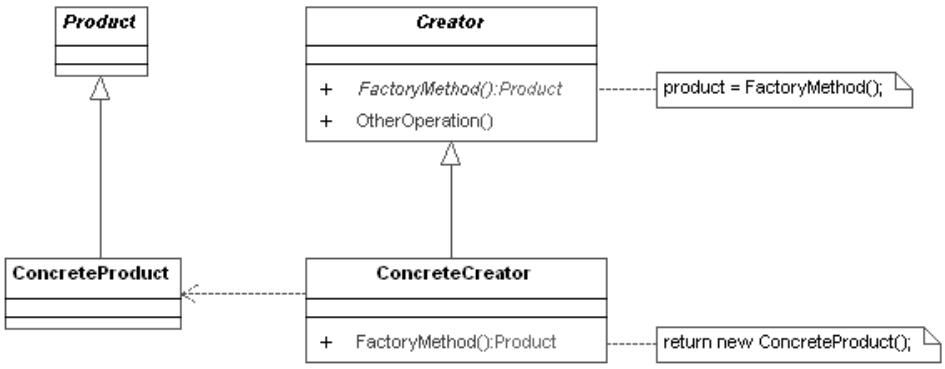


Figura 7.1. *Diagrama de clase a şablonului Metoda Fabrică*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

O altă diagramă în care avem doi creatori şi două produse este prezentată în figura 7.2.

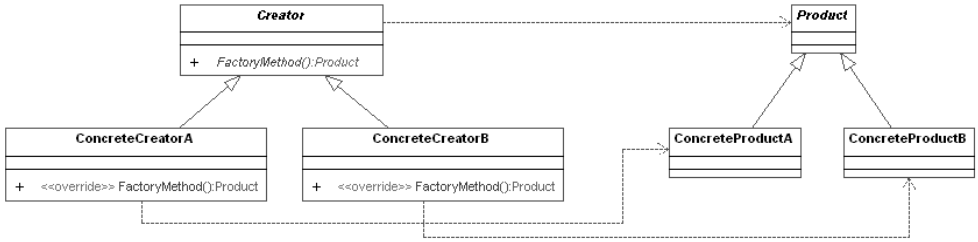


Figura 7.2. *Diagrama de clase cu doi creatori şi două produse*

Metoda Fabrică se foloseşte în general atunci când o clasă nu poate şti sau nu doreşte să specifice din ce clasă va fi creat un obiect şi în consecinţă lasă clasele derivate să specifice clasa acestuia.

3. Exemplu de implementare

Codul C# corespunzător diagramei UML anterioare este prezentat mai jos.

Clasele Produs

```
abstract class Product
{
}

class ConcreteProductA : Product
{
}

class ConcreteProductB : Product
{
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Clasele Creator

```
abstract class Creator
{
   // Metode
   abstract public Product FactoryMethod();
}


class ConcreteCreatorA : Creator
{
   // Metode
   override public Product FactoryMethod()
   {
      return new ConcreteProductA();
   }
}

class ConcreteCreatorB : Creator
{
   // Metode
   override public Product FactoryMethod()
   {
      return new ConcreteProductB();
   }
}
```

Clientul

```
class Client
{
   public static void Main( string[] args )
   {
      // FactoryMethod returnează ProductA
      Creator c = new ConcreteCreatorA();
      Product p = c.FactoryMethod();
      Console.WriteLine("A fost creat {0}", p );

      // FactoryMethod returnează ProductB
      c = new ConcreteCreatorB();
      p = c.FactoryMethod();
      Console.WriteLine("A fost creat {0}", p );
   }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

4. Moştenirea şi polimorfismul

4.1. Polimorfismul

Cu ajutorul moştenirii, o clasă poate fi folosită ca şi cum ar reprezenta mai multe tipuri. Ea poate fi folosită ca propriul său tip, ca un tip de clasă de bază, sau un tip de interfață pe care o implementează. Acest comportament este numit polimorfism. În C#, orice tip este polimorfic. Un tip poate fi folosit aşa cum este sau ca o instanță de object, pentru că orice tip consideră automat tipul object ca tip de bază.

Polimorfismul este important nu numai pentru clasele derivate, ci şi pentru clasele de bază. Când se foloseşte o clasă de bază, se poate folosi de fapt orice clasă derivată. Proiectanții unei clase de bază pot anticipa aspectele claselor care se vor modifica prin derivare. De exemplu, o clasă de bază pentru maşini poate conține membri care se modifică dacă maşina este o dubiță sau o maşină decapotabilă. O clasă de bază poate marca acei membri ca virtuali, dând posibilitatea claselor derivate care reprezintă maşina decapotabilă şi dubița să suprascrie comportamentul respectiv.

Când o clasă derivată moşteneşte o clasă de bază, ea primeşte toate metodele, câmpurile, proprietățile şi evenimentele clasei de bază. Pentru a modifica datele şi metodele unei clase de bază există două posibilități: se poate înlocui clasa de bază cu una derivată sau se poate suprascrie un membru virtual din clasa de bază.

4.2. Clase abstracte

Cuvântul cheie abstract oferă posibilitatea creării claselor şi membrilor de clase doar pentru a putea fi moşteniți: pentru a defini trăsături ale claselor derivate, neabstracte.

```
public abstract class A
{
   // membrii clasei
}
```

O clasă abstractă nu poate fi instanțiată. Scopul său este să ofere o definiție comună pentru mai multe clase derivate. De exemplu, o bibliotecă de clase poate defini o clasă abstractă care este folosită ca parametru în mai

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

multe funcţii din bibliotecă şi necesită ca dezvoltatorii care o folosesc să asigure o implementare proprie prin crearea unei clase derivate.

Clasele abstracte pot de asemenea să definească metode abstracte, prin adăugarea cuvântului cheie abstract în faţa tipului returnat de metodă. De exemplu:

```
public abstract class A
{
    public abstract void Method(int i);
}
```

Metodele abstracte nu au implementare şi deci definiţia metodei este urmată de „ ; ” în locul unui bloc normal de cod. Clasele derivate neabstracte trebuie să implementeze toate metodele abstracte.

4.3. Interfeţe

Interfeţele sunt definite cu ajutorul cuvântului cheie interface. De exemplu:

```
interface IComparable
{
    int CompareTo(object obj);
}
```

Interfeţele descriu un grup de funcţionalităţi asemănătoare care pot să aparţină oricărei clase sau structuri. Interfeţele pot conţine metode, proprietăţi, evenimente sau indecşi, însă nu pot conţine câmpuri.

Membrii unei interfeţe sunt implicit publici.

Clasele şi structurile pot moşteni interfeţe într-un mod asemănător claselor care moştenesc clase sau structuri, cu două excepţii:

- O clasă sau structură poate moşteni mai multe interfeţe;
- Când o clasă sau structură moşteneşte o interfaţă, moşteneşte doar numele metodelor şi semnăturile acestora, deoarece interfeţele nu conţin implementări.

În exemplul următor, clasa Minivan este derivată din clasa Car şi implementează interfaţa IComparable.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
public class Minivan : Car, IComparable
{
   public int CompareTo(object obj)
   {
         // implementarea metodei CompareTo
         return 0;  // dacă obiectele sunt egale
   }
}
```

Pentru a implementa un membru al unei interfețe, membrul respectiv trebuie să fie public, nestatic şi să aibă acelaşi nume şi semnătură cu membrul interfeței.

Interfețele pot moşteni alte interfețe. Este posibil ca o clasă să moştenească o interfață de mai multe ori, printr-o clasă sau interfață moştenită. În acest caz, clasa poate implementa interfața doar o dată.

O interfață are următoarele caracteristici:

- Este similară unei clase de bază abstracte: orice tip neabstract care moşteneşte o interfață trebuie să îi implementeze toți membrii;
- O interfață nu poate fi instanțiată direct;
- Interfețele pot conține metode, proprietăți, evenimente şi indecşi;
- Interfețele nu conțin implementări ale metodelor;
- Clasele şi structurile pot moşteni mai multe interfețe;
- O interfață poate ea însăşi să moştenească mai multe interfețe.

4.4. Membri virtuali

Pentru ca o clasă derivată să reimplementeze un membru al unei clase de bază, clasa de bază trebuie să definească membrul ca virtual iar clasa derivată să folosească cuvântul cheie override pentru a înlocui implementarea membrului. De exemplu:

```
public class BaseClass
{
   public virtual void Method()
   {
      ...
   }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    public virtual int Property
    {
        get { ... }
    }
}

public class DerivedClass : BaseClass
{
    public override void Method()
    {
        ...
    }

    public override int Property
    {
        get { ... }
    }
}
```

Câmpurile nu pot fi virtuale, doar metodele, proprietăţile şi indecşii. Când o clasă derivată suprascrie un membru virtual, acel membru este apelat chiar şi atunci când o instanţă a acelei clase este accesată ca o instanţă a clasei de bază. De exemplu:

```
DerivedClass B = new DerivedClass();
B.Method (); // Apelează metoda nouă

BaseClass A = (BaseClass)B;
A.Method (); // Apelează tot metoda nouă
```

Un membru virtual rămâne astfel în toată ierarhia de clase, indiferent de numărul de niveluri dintre clasa de bază şi cea curentă. Dacă în clasa `A` este declarat un membru virtual şi clasa `B` este derivată din clasa `A` iar clasa `C` este derivată din clasa `B`, atunci clasa `C` moşteneşte membrul virtual şi poate să îl suprascrie, chiar dacă în clasa `B` el a fost deja suprascris. De exemplu:

```
public class A
{
    public virtual void Method()
    {
        ...
    }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
public class B : A
{
    public override void Method()
    {
        ...
    }
}

public class C : B
{
    public override void Method()
    {
        ...
    }
}
```

4.5. Clase sigilate şi membri sigilaţi

Sigilarea împiedică moştenirea claselor sau suprascrierea membrilor virtuali.

O clasă pot fi declarată ca sigilată, punând cuvântul cheie sealed înaintea cuvântului class la definirea acesteia:

```
public sealed class A
{
    // membrii clasei
}
```

O clasă sigilată nu poate fi folosită drept clasă de bază. Din această cauză, ea nu poate fi nici abstractă. Clasele sigilate sunt folosite, în principiu, pentru a preveni derivarea. Neputând fi folosite ca şi clase de bază, unele optimizări în timp real pot face apelul membrilor acestora mai rapid.

Un membru, o metodă, o proprietate sau un eveniment al unei clase derivate care suprascrie un membru virtual al clasei de bază poate declara acel membru ca sigilat. Astfel se elimină aspectul virtual al membrului pentru orice clasă viitoare derivată. De exemplu:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
public class B
{
    public virtual void Method()
    {
        ...
    }
}

public class C : B
{
    public sealed override void Method()
    {
        ...
    }
}
```

În exemplul anterior, metoda Method nu mai este virtuală pentru nicio clasă derivată din C. Dar este încă virtuală pentru instanțele clasei C.

4.6. Înlocuirea unui membru cu ajutorul cuvântului cheie *new*

Înlocuirea unui membru din clasa de bază cu unul nou, derivat, necesită folosirea cuvântului cheie new. Dacă o clasă de bază defineşte o metodă, un câmp sau o proprietate, cuvântul cheie new se utilizează pentru a crea o nouă definiție în clasa derivată. Cuvântul cheie new trebuie pus înaintea tipului returnat al membrului clasei. De exemplu:

```
public class BaseClass
{
    public void Method()
    {
        ...
    }

    public int Property
    {
        get { ... }
    }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
public class DerivedClass : BaseClass
{
    public new void Method()
    {
        ...
    }

    public new int Property
    {
        get { ... }
    }
}
```

Când se foloseşte cuvântul cheie new, noul membru este apelat în locul celui vechi, înlocuit. Membrii respectivi ai clasei de bază sunt numiți „membri ascunşi”. Ei mai pot fi apelați doar dacă o instanță a clasei derivate este convertită prin *cast* la o instanță a clasei de bază. De exemplu:

```
DerivedClass B = new DerivedClass();
B.Method (); // Apelează metoda nouă

BaseClass A = (BaseClass)B;
A.Method (); // Apelează metoda veche
```

4.7. Accesarea clasei de bază cu ajutorul cuvântului cheie *base*

O clasă derivată care a înlocuit sau suprascris o metodă sau proprietate poate încă accesa metoda sau proprietatea din clasa de bază folosind cuvântul cheie base. De exemplu:

```
public class A
{
    public virtual void Method()
    {
        ...
    }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
public class B : A
{
    public override void Method()
    {
        ...
    }
}

public class C : B
{
    public override void Method()
    {
        // apelează Method din B pentru a utiliza comportamentul lui B
        base.Method();

        // comportamentul specific lui C urmează în continuare
        ...
    }
}
```

Se recomandă ca un membru virtual să utilizeze cuvântul cheie base pentru a apela implementarea membrului din clasa de bază în implementarea proprie. Prin aceasta este lăsată clasa derivată să se axeze pe implementarea propriilor funcționalități. Dacă nu este apelată implementarea din clasa de bază, atunci clasa derivată va trebui să se ocupe de implementarea unor funcționalități similare cu acelea ale clasei de bază.

Cuvântul cheie base este folosit:

- Pentru a apela o metodă din clasa de bază care a fost suprascrisă de o altă metodă;
- Pentru a specifica clasa de bază al cărei constructor să fie apelat la instanțierea clasei derivate.

Din metodele statice nu este permis accesul la clasa de bază prin cuvântul cheie base.

5. Aplicaţii

5.1. Realizați o aplicație care deschide şi afişează fişiere text şi grafice (figura 7.3). Testul se va efectua după extensia fişierului (*txt, rtf, bmp, jpg*). Se va utiliza şablonul de proiectare *Metoda fabrică*.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

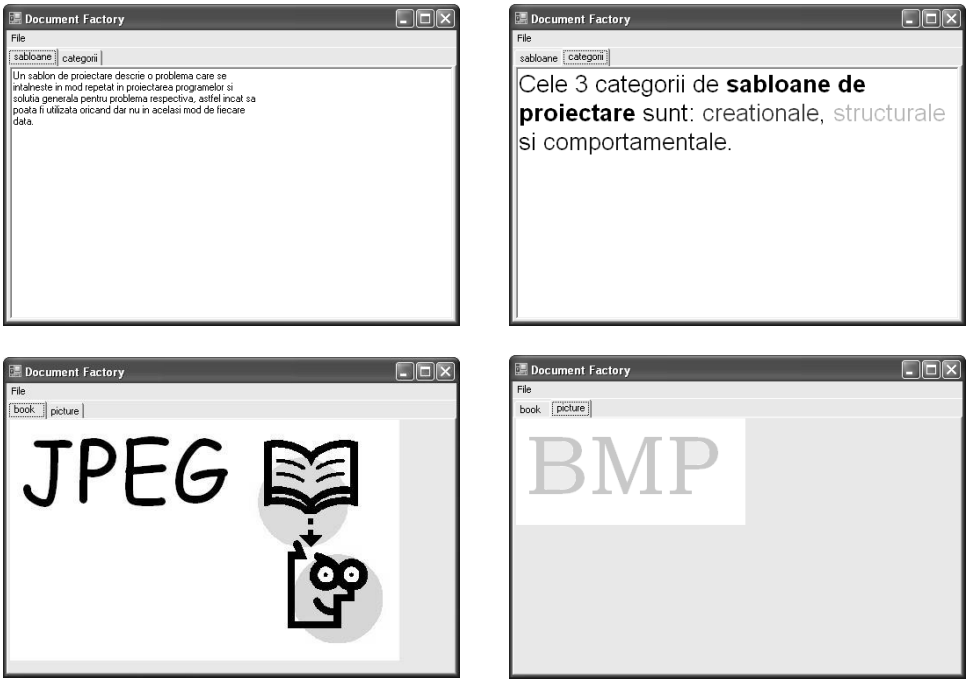


Figura 7.3. *Exemplu de rezolvare: interfața cu utilizatorul*

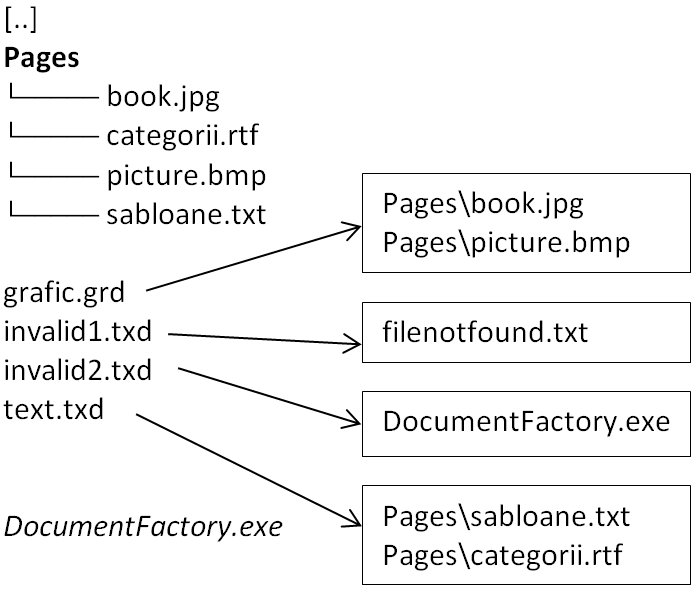


Figura 7.4. *Exemplu de rezolvare: fișierele de test*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Aplicaţia deschide fişiere „index” cu extensiile *txd*, pentru fişiere text, respectiv *grd*, pentru fişiere grafice. Aceste fişiere index conţin căile către mai multe fişiere cu extensiile *txt* sau *rtf*, respectiv *bmp* sau *jpg*.

Un exemplu privind structura de directoare şi fişiere, precum şi conţinutul fişierelor index este prezentat în figura 7.4.

Diagrama de clase este prezentată în figura 7.5.

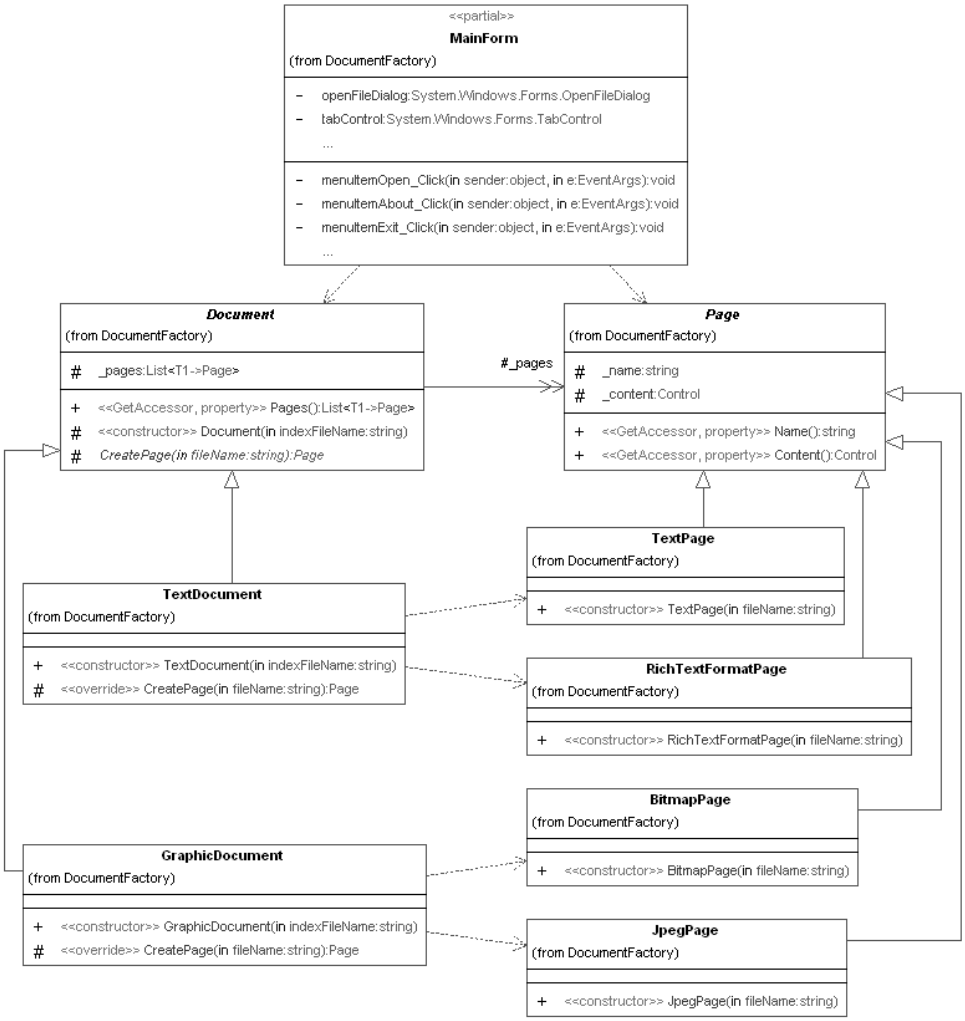


Figura 7.5. *Exemplu de rezolvare: diagrama de clase*

Indicaţii. Pentru citirea fişierelor text se poate utiliza clasa StreamReader din System.IO. Pentru fişierele text şi *rtf* se poate utiliza clasa RichTextBox din System.Windows.Forms cu metoda LoadFile. Pentru

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

fişierele grafice se poate utiliza clasa Bitmap unde numele fişierului este primit în constructor sau proprietatea ImageLocation din clasa PictureBox.

Pentru deschiderea unui fişier se utilizează un control OpenFileDialog. Pentru a se deschide doar tipurile de fişiere cu extensiile *txd* sau *grd*, se setează proprietatea Filter cu valoarea "Text documents (*.txd)|*.txd|Graphic documents (*.grd)|*.grd".

În evenimentul corespunzător butonului *Open*, se poate prelucra un fişier doar dacă utilizatorul a ales un fişier şi a dat *OK* în fereastra de deschidere de fişiere.

```
if (openFileDialog.ShowDialog() != DialogResult.OK)
    return;
```

Apoi se testează valoarea proprietății FilterIndex din obiectul openFileDialog, care indică tipul de fişier deschis (index text sau index grafic). Indexul pleacă de la 1, nu de la 0. În continuare, se instanțiază un obiect TextDocument sau un GraphicDocument. Afişarea paginilor în obiectul tabControl se poate face în modul următor:

```
Document doc;

... // crearea obiectului document (creatorul concret)

tabControl.Controls.Clear();
foreach (Page p in doc.Pages)
{
    TabPage tp = new TabPage(p.Name);
    p.Content.Dock = DockStyle.Fill;
    tp.Controls.Add(p.Content);
    tabControl.TabPages.Add(tp);
}
```

Secvența de cod de mai sus se recomandă a fi inclusă într-un bloc de tratare a excepțiilor, deoarece pot apărea probleme referitoare la deschiderea unui fişier care să nu conțină căi valide sau fişierele indicate de aceste căi să nu aibă conținutul aşteptat.

În produsele concrete se creează controalele corespunzătoare iar numele paginilor afişate în tabControl se determină din numele fişierului încărcat:

```
_name = Path.GetFileNameWithoutExtension(fileName);
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

În constructorul creatorului abstract (Document) se citeşte fişierul index linie cu linie şi se creează paginile corespunzătoare:

```
StreamReader sr = new StreamReader(indexFileName);
string line;
while ((line = sr.ReadLine()) != null)
{
    if (line != string.Empty)
        _pages.Add(CreatePage(line));
}
sr.Close();
```

Însă deoarece metoda CreatePage este abstractă, deciziile privind tipul efectiv al paginii vor fi luate de clasele derivate din Document.

5.2. *Temă pentru acasă*. Adăugați în proiect un nou tip de pagină care să afişeze un document Microsoft Word şi/sau un document PDF.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 8

Şabloanele de proiectare Singleton şi Prototip

1. Obiective
2. Şablonul creaţional *Singleton*
3. Şablonul creaţional *Prototip*
4. Aplicaţii

1. Obiective

Obiectivele capitolului 8 sunt următoarele:

- Implementarea unui program după şablonul de proiectare *Singleton*;
- Implementarea unui program după şablonul de proiectare *Prototip* (engl. "Prototype").

2. Şablonul creaţional *Singleton*

Scopul şablonului *Singleton* este să garanteze faptul că o clasă poate avea doar o singură instanţă şi să asigure un punct global de acces la ea.

În unele situaţii, este important ca o clasă să aibă o singură instanţă, de exemplu atunci când avem dispozitive hardware sau accesorii ataşate unui calculator şi dorim să prevenim accesul concurent la acestea. Alte situaţii sunt cele în care se doreşte lucrul cu registrul Windows (unic) sau atunci când se lucrează cu un bazin (engl. "pool") de fire de execuţie. În general, şablonul este util când o resursă unică trebuie să aibă un corespondent unic care o accesează din program.

Este nevoie deci de o modalitate de a împiedica instanţierile multiple ale unei clase şi de a asigura o metodă unică globală pentru accesarea instanţei. Avantajul faţă de utilizarea unor clase statice sau cu proprietăţi statice este faptul că *Singleton*-ul poate fi derivat şi astfel clienţii îi pot extinde funcţionalitatea fără a fi nevoiţi să modifice clasa existentă.

Diagrama de clase este prezentată în figura 8.1.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

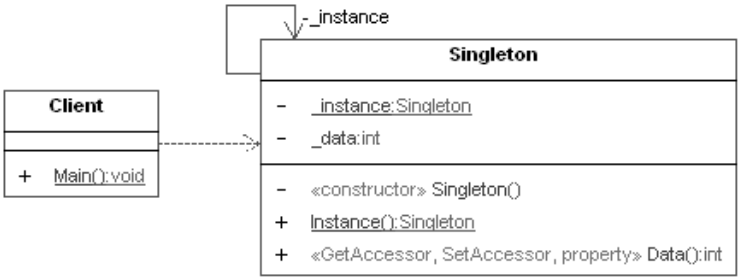


Figura 8.1. *Diagrama de clase a şablonului Singleton*

Clientul poate accesa poate accesa instanța unică a clasei Singleton utilizând metoda statică Instance.

2.1. Exemplu de implementare

Codul C# corespunzător diagramei UML anterioare este prezentat mai jos.

Clasa Singleton

```
class Singleton
{
   private static Singleton _instance;
   private int _data;  // datele propriu-zise ale clasei
   public int Data
   {
      get { return _data; }
      set { _data = value; }
   }
   private Singleton() // protected dacă avem nevoie de clase derivate
   {
   }
   public static Singleton Instance()
   {
      // Iniţializare întârziată ("lazy initialization")
      if( _instance == null )
         _instance = new Singleton();
      return _instance;
   }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Clientul

```
public class Client
{
    public static void Main()
    {
        // constructorul este privat, nu se poate folosi "new"
        Singleton s1 = Singleton.Instance();
        s1.Data = 5;

        Singleton s2 = Singleton.Instance();
        Console.WriteLine("Rezultat: {0}", s2.Data);    // "Rezultat: 5"

        Console.ReadLine();
    }
}
```

Trebuie precizat că în situațiile în care mai multe fire de execuție pot accesa simultan secțiunea de inițializare, trebuie incluse mecanisme suplimentare de sincronizare.

3. Şablonul creaţional *Prototip*

Scopul şablonului *Prototip* este să specifice tipurile de obiecte care pot fi create folosind o instanță prototip şi să creeze noi obiecte prin copierea acestui prototip.

Există situații în care instanțierea unui obiect în mod normal, folosind inițializarea stării interne în constructor, este costisitoare din punct de vedere al timpului sau resurselor de calcul. De aceea, dacă este nevoie de mai multe astfel de obiecte în sistem, se poate utiliza un obiect prototip pentru a crea noi obiecte, prin copierea, în loc de calcularea de fiecare dată a valorilor datelor interne. Ideea de bază este utilizarea unei instanțe tipice pentru a crea o altă instanță înrudită.

Şablonul foloseşte o metodă caracteristică numită Clone pentru crearea cópiilor unui obiect. Nu se specifică însă dacă clonarea este *superficială* (engl. "shallow") sau *profundă* (engl. "deep"), aceasta depinde de tipul datelor copiate. Clonarea superficială este mai simplă şi se foloseşte de obicei când câmpurile sunt de tip valoare (tipuri primitive, structuri). Pentru tipuri referință, acest tip de clonare copiază doar referințele şi de aceea, în aceste cazuri se poate prefera copierea profundă, care copiază recursiv toate valorile.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

De exemplu, dacă obiectul are două câmpuri primitive, int şi double, este suficientă clonarea superficială. Dacă se mai adaugă un câmp vector, int[], clonarea superficială ar copia doar referința: obiectul prototip şi copia ar referenția de fapt acelaşi vector. Clonarea profundă creează în acest caz doi vectori distincți pentru cele două obiecte.

Una din principalele probleme ale şablonului este legată de tipul clonării, deoarece varianta recomandată depinde de situație.

Diagrama de clase este cea din figura 8.2.

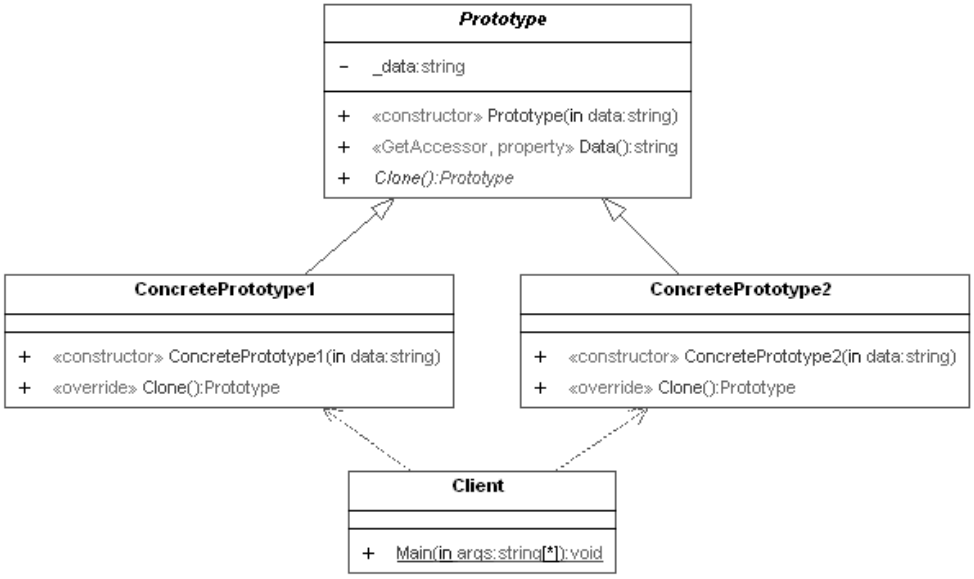


Figura 8.2. *Diagrama de clase a şablonului Prototip*

3.1. Exemplu de implementare

Codul C# corespunzător diagramei UML anterioare este prezentat mai jos.

Prototipul abstract

```
abstract class Prototype
{
   private string _data;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
   public Prototype(string data)
   {
      _data = data;
   }

   public string Data
   {
      get
      {
         return _data;
      }
   }

   abstract public Prototype Clone();
}
```

Prototipurile concrete

```
class ConcretePrototype1 : Prototype
{
   public ConcretePrototype1(string data) : base(data)
   {
   }

   override public Prototype Clone()
   {
      // copie superficială
      return (Prototype)this.MemberwiseClone();
   }
}


class ConcretePrototype2 : Prototype
{
   public ConcretePrototype2(string data) : base(data)
   {
   }

   override public Prototype Clone()
   {
      // copie superficială
      return (Prototype)this.MemberwiseClone();
   }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Clientul

```
class Client
{
   public static void Main(string[] args)
   {
      // se creează două prototipuri şi se clonează fiecare

      ConcretePrototype1 prototype1 = new ConcretePrototype1("Proto1");
      ConcretePrototype1 clone1 = (ConcretePrototype1)p1.Clone();
      Console.WriteLine("Cloned: {0}", c1.Data);

      ConcretePrototype2 prototype2 = new ConcretePrototype2("Proto2");
      ConcretePrototype2 clone2 = (ConcretePrototype2)p2.Clone();
      Console.WriteLine("Cloned: {0}", c2.Data);

      Console.ReadLine();
   }
}
```

4. Aplicaţii

4.1. Să se simuleze lucrul cu o imprimantă (figura 8.3). Fiecărui document îi este asociată o mărime. Imprimanta are o coadă de documente ce urmează a fi tipărite. Când se trimite un document la imprimare, dacă această coadă este vidă, fişierul începe să fie tipărit. Timpul necesar tipăririi este proporțional cu mărimea sa. Când coada imprimantei nu este vidă, documentul este doar introdus în coadă. La terminarea tipăririi unui document, se preia următorul din coadă (dacă există). Implementarea se va realiza utilizând şablonul *Singleton*.

Diagrama de clase a aplicației este prezentată în figura 8.4.

Indicație: pentru simularea tipăririi se recomandă folosirea unui control Timer, descris în capitolul 2, secțiunea 5. Evenimentul Tick este tratat o dată la un interval de timp, specificat în milisecunde de proprietatea Interval. De exemplu, dacă Interval = 500, codul evenimentului Tick va fi executat în mod repetat, de două ori pe secundă. Proprietatea Enabled arată dacă *timer*-ul e activat sau nu (true/false).

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

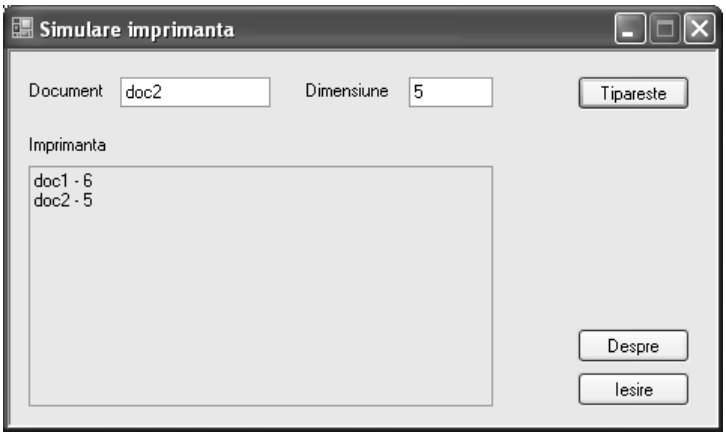


Figura 8.3. *Exemplu de rezolvare: interfaţa cu utilizatorul*

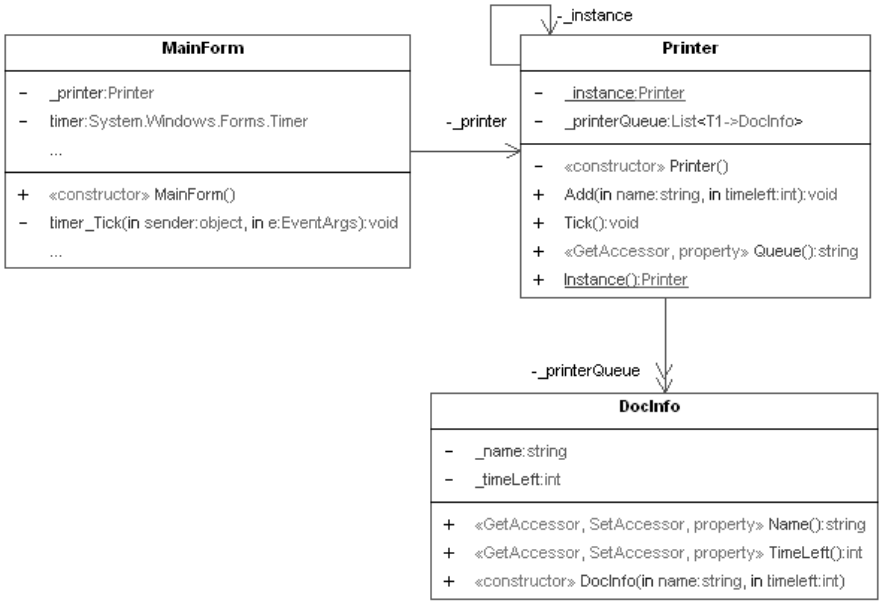


Figura 8.4. *Exemplu de rezolvare: diagrama de clase*

Proprietatea Queue din clasa Printer returnează conţinutul cozii de activităţi ale imprimantei, pentru a fi afişată ca atare de client (MainForm). În acest fel, clientul nu mai trebuie să depindă de clasa DocInfo, care reprezintă obiectele cu care lucrează intern doar clasa Printer.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

4.2. Să presupunem că avem un joc în care utilizatorul împuşcă monştri pe ecran (figura 8.5). Există mai multe tipuri de monştri, fiecare cu propriile caracteristici: imagine, culoare, număr de vieți etc. Pe lângă acestea, fiecare monstru are implementat un modul de inteligență artificială, a cărui inițializare necesită multe resurse.

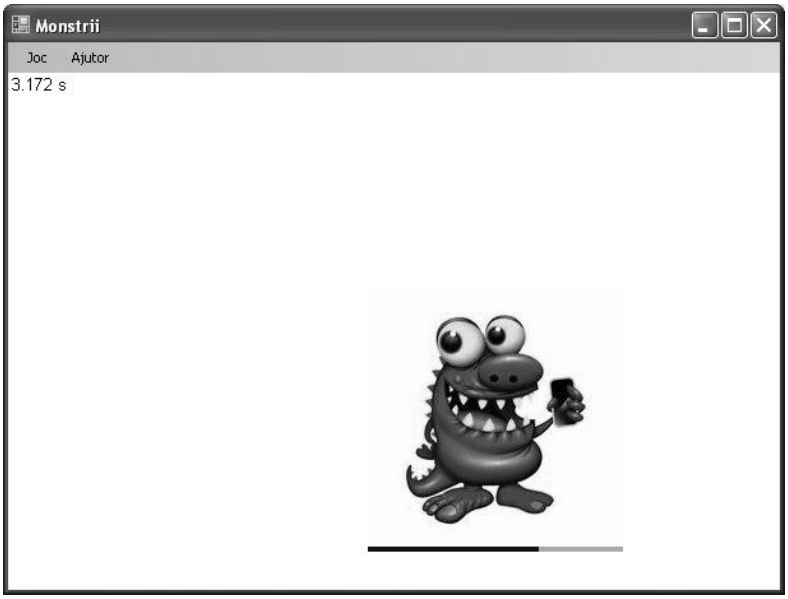


Figura 8.5. *Exemplu de rezolvare: interfața cu utilizatorul*

Scheletul programului este dat, la fel şi o clasă pentru un monstru implementată în *MonsterSprite.dll*, cu structura din diagrama din figura 8.6 şi al cărei cod sursă este prezentat în continuare.

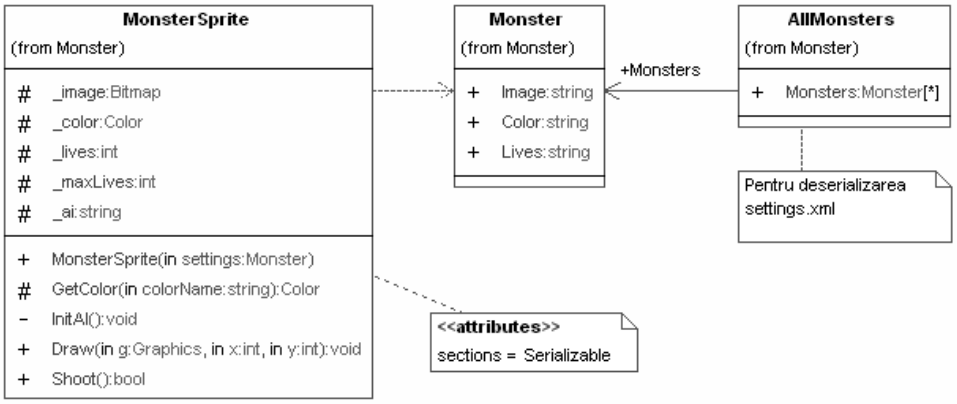


Figura 8.6. *Diagrama claselor din DLL*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

MonsterSprite.cs

```
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;
using System.Runtime.Serialization;

namespace Monster
{
   [Serializable]
   public class MonsterSprite
   {
      protected Bitmap _image;
      protected Color _color;
      protected int _lives;
      protected int _maxLives;
      protected string _ai = "I am stupid";

      public MonsterSprite(Monster settings)
      {
         _image = new Bitmap(settings.Image);
         _maxLives = Convert.ToInt32(settings.Lives);
         _lives = _maxLives;
         _color = GetColor(settings.Color);

         InitAI();
      }

      /// <summary>
      /// Interpretează numele unei culori şi returnează un obiect Color
      /// </summary>
      /// <param name="colorName"></param>
      /// <returns></returns>
      protected Color GetColor(string colorName)
      {
         Color c;
         switch (colorName)
         {
            case "red": c = Color.Red; break;
            case "blue": c = Color.Blue; break;
            case "green": c = Color.Green; break;
            case "yellow": c = Color.Yellow; break;
            case "orange": c = Color.Orange; break;
            case "black": c = Color.Black; break;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
            default: c = Color.Gray; break;
        }
        return c;
    }

    private void InitAI()
    {
        _ai = "I am smart";
        MessageBox.Show("Se initializeaza modulul de inteligenta artificiala...");
        Thread.Sleep(2000);
    }

    /// <summary>
    /// Desenează un personaj
    /// </summary>
    /// <param name="g"></param>
    /// <param name="x"></param>
    /// <param name="y"></param>
    public void Draw(Graphics g, int x, int y)
    {
        g.DrawImage(_image, x, y);
        double more = (double)_lives / (double)_maxLives;
        g.FillRectangle(new SolidBrush(Color.DarkGray), (float)x, (float)(y + 200),
            (float)200, (float)4);
        g.FillRectangle(new SolidBrush(_color), (float)x, (float)(y + 200),
            (float)(more * 200), (float)4);
    }

    /// <summary>
    /// Returnează true dacă este mort
    /// </summary>
    /// <returns></returns>
    public bool Shoot()
    {
        _lives--;
        if (_lives == 0)
            return true;
        else
            return false;
    }
  }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Utils.cs

```
namespace Monster
{
   public class AllMonsters
   {
      public Monster[] Monsters;
   }

   public class Monster
   {
      public string Image;
      public string Color;
      public string Lives;
   }
}
```

Scopul aplicației este să evitați inițializarea de fiecare dată a modulului costisitor de IA (metoda InitAI), înlocuind instanțierea unui obiect MonsterSprite cu clonarea sa şi modificarea caracteristicilor variabile. În acest caz, va exista un PrototypeManager care va asigura crearea monştrilor, înlocuind o instrucțiune de tipul:

```
_mainMonster = new MonsterSprite(_listMonsters[_monsterType]);
```

cu:

```
_mainMonster = _prototypeManager.GetMonster(_monsterType);
```

DLL-ul conținând clasa MonsterSprite se va folosi ca atare, fără modificări.

Indicație: pentru a face rapid o copie profundă a unui obiect, se poate utiliza serializarea într-o metodă de tipul:

```
public static object Clone(object obj)
{
   object objClone = null;
   MemoryStream memory = new MemoryStream();
   BinaryFormatter binForm = new BinaryFormatter();
   binForm.Serialize(memory, obj);
   memory.Position = 0;
   objClone = binForm.Deserialize(memory);
   return objClone;
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

În vederea utilizării acestei modalități de copiere, trebuie incluse în proiect *namespace*-urile System.Runtime.Serialization și System.IO, iar clasa trebuie decorată cu atributul [Serializable]. Această metodă generală trebuie particularizată la situația concretă a aplicației.

În continuare, se prezintă scheletul programului, care trebuie optimizat după cerințele precizate mai sus.

MainForm.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
using System.Xml;
using System.Xml.Serialization;
using System.IO;

namespace Monster
{
   public partial class MainForm : Form
   {
      List<Monster> _listMonsters;
      MonsterSprite _mainMonster;
      Random _rand = new Random();
      int _monsterType = 0;
      int _x, _y;
      long _elapsed;
      const int MonsterSize = 200;
      const int MaxLevels = 4;
      bool _gameOver = false;

      public MainForm()
      {
         InitializeComponent();
      }

      private void loadSettingsToolStripMenuItem_Click(object sender, EventArgs e)
      {
         // încarcă
         try
         {
            XmlSerializer serializer = new XmlSerializer(typeof(AllMonsters));
            FileStream fs = new FileStream("settings.xml", FileMode.Open);
            XmlReader reader = new XmlTextReader(fs);
            AllMonsters ab = (AllMonsters)serializer.Deserialize(reader);
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
            reader.Close(); fs.Close();
            serializer = null;

            _listMonsters = new List<Monster>();

            for (int i = 0; i < ab.Monsters.Length; i++)
               _listMonsters.Add(ab.Monsters[i]);
         }
         catch
         {
            MessageBox.Show("Nu s-a putut incarca settings.xml");
            return;
         }

         if (_listMonsters == null || _listMonsters.Count == 0)
         {
            MessageBox.Show("Fisier de configurare invalid: settings.xml");
            _listMonsters = null;
            return;
         }
      }

      private void startNewGameToolStripMenuItem_Click(object sender, EventArgs e)
      {
         // start
         if (_listMonsters == null || _listMonsters.Count == 0)
            loadSettingsToolStripMenuItem.PerformClick();
         if (_listMonsters == null)
            return;

         _monsterType = 0;
         _gameOver = false;

         try
         {
            _mainMonster = new MonsterSprite(_listMonsters[_monsterType]);
         }
         catch (Exception exc)
         {
            MessageBox.Show(exc.Message);
            _mainMonster = null;
            return;
         }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    _elapsed = DateTime.Now.Ticks;
    timer.Start();

    Redraw();
  }

  private void Redraw()
  {
    try
    {
      _x = _rand.Next(pictureBox.Width - MonsterSize + 20);
      _y = _rand.Next(pictureBox.Height - MonsterSize - 10);
      pictureBox.Refresh();
    }
    catch
    {
      MessageBox.Show("Fereastra este prea mica");
      _mainMonster = null;
      return;
    }
  }

  private void exitToolStripMenuItem_Click(object sender, EventArgs e)
  {
    Close();
  }

  private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
  {
    // despre program

    const string copyright =
      "Sablonul de proiectare Prototip\r\n" +
      "Ingineria programarii\r\n" +
      "(c) 2008-2012 Florin Leon\r\n" +
      " http://florinleon.byethost24.com/lab_ip.htm ";

    MessageBox.Show(copyright, "Despre Monstri");
  }

  private void pictureBox_Paint(object sender, PaintEventArgs e)
  {

    if (_mainMonster == null)
    {
      timer.Stop();
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
            e.Graphics.Clear(Color.White);
            if (_gameOver)
            {
               e.Graphics.DrawString("Jocul s-a terminat!", new Font("Arial", 48),
                  Brushes.Red, 10, 10);
               long dt = DateTime.Now.Ticks - _elapsed;
               double ms = dt / 10000000.0;
               e.Graphics.DrawString(ms.ToString("F3") + " s", new Font("Arial", 48),
                  Brushes.Red, 10, 80);
            }
            return;
         }

         _mainMonster.Draw(e.Graphics, _x, _y);
      }

      private void ShootMonster()
      {
         if (_mainMonster.Shoot())
         {
            _monsterType++;
            if (_monsterType < MaxLevels)
            {
               try
               {
                  _mainMonster = new MonsterSprite(_listMonsters[_monsterType]);
               }
               catch (Exception exc)
               {
                  MessageBox.Show(exc.Message);
                  _mainMonster = null;
                  return;
               }

               Redraw();
            }
            else
            {
               _mainMonster = null;
               _gameOver = true;
               pictureBox.Refresh();
            }
         }
         else
            Redraw();
      }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    private void pictureBox_MouseDown(object sender, MouseEventArgs e)
    {
      if (_mainMonster != null)
      {
        if (e.X > _x && e.X < _x + MonsterSize && e.Y > _y && e.Y < _y + MonsterSize)
          ShootMonster();
      }
    }

    private void timer_Tick(object sender, EventArgs e)
    {
      Graphics g = pictureBox.CreateGraphics();
      long dt = DateTime.Now.Ticks - _elapsed;
      double ms = dt / 10000000.0;
      g.FillRectangle(Brushes.White, 1, 1, 100, 20);
      g.DrawString(ms.ToString("F3") + " s", new Font("Arial", 10), Brushes.Black, 1, 1);
    }

    private void Form1_Load(object sender, EventArgs e)
    {
      this.WindowState = FormWindowState.Maximized;
    }
  }
}
```

Un exemplu de fişier cu setări este dat mai jos.

settings.xml

```
<?xml version="1.0" encoding="us-ascii"?>
<AllMonsters>
 <Monsters>
  <Monster>
   <Image>monster1.jpg</Image>
   <Color>blue</Color>
   <Lives>3</Lives>
  </Monster>
  <Monster>
   <Image>monster2.jpg</Image>
   <Color>green</Color>
   <Lives>3</Lives>
  </Monster>
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    <Monster>
      <Image>monster3.jpg</Image>
      <Color>black</Color>
      <Lives>4</Lives>
    </Monster>
    <Monster>
      <Image>monster4.jpg</Image>
      <Color>red</Color>
      <Lives>5</Lives>
    </Monster>
  </Monsters>
</AllMonsters>
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 9

Şablonul de proiectare Faţadă

1. Obiectiv
2. Scop şi motivaţie
3. Aplicabilitate
4. Analiza şablonului
5. Exemplu de implementare
6. Aplicaţie

1. Obiectiv

Obiectivul capitolului 9 este implementarea unui program după şablonul de proiectare structural *Fațadă* (engl. "Façade").

2. Scop şi motivaţie

Şablonul *Fațadă* asigură o interfață unitară pentru o mulțime de interfețe ale unui subsistem. Fațada defineşte o interfață de nivel înalt care face subsistemul mai uşor de utilizat.

Structurarea unui sistem în subsisteme reduce complexitatea. Unul din scopurile unei proiectări corecte este acela de a minimiza comunicațiile şi dependențele dintre subsisteme. Un mod de a obține aceasta este introducerea fațadei, obiect care furnizează o singură interfață simplificată pentru funcționalitățile mai generale ale subsistemului.

De exemplu, putem considera un mediu de programare care dă aplicațiilor acces către un subsistem de compilare ce conține clase precum Scanner, Parser, ProgramNode, BytecodeStream şi ProgramNodeBuilder (figura 9.1). Unele aplicații specializate ar putea avea nevoie să acceseze aceste clase direct, dar majoritatea aplicațiilor care folosesc compilatorul nu sunt interesate de detalii precum parsarea sau generarea de cod, ci doresc doar să compileze un program. Interfețele puternice dar de nivel scăzut din subsistemul compilator le complică sarcina.

Pentru a asigura o interfață de nivel înalt care să separe clienții de aceste clase, subsistemul compilator include de asemenea o clasă Compiler. Această clasă defineşte o interfață unitară pentru funcționalitatea

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

compilatorului şi se comportă ca o fațadă, oferind clientului o interfață simplificată pentru subsistemul compilator.

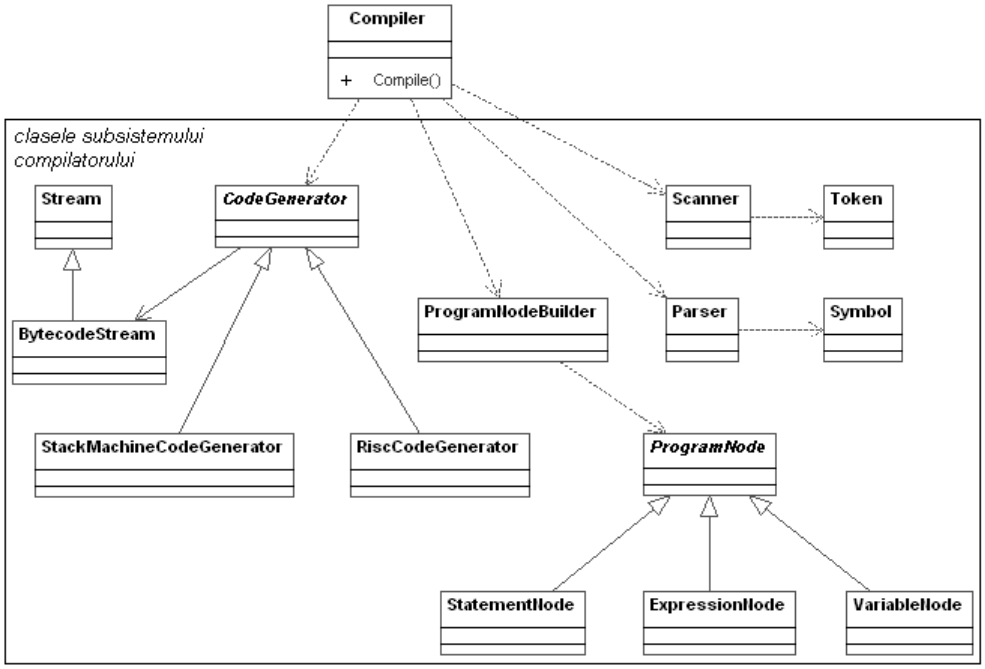


Figura 9.1. *Exemplu de sistem complex apelabil prin intermediul unei Fațade*

3. Aplicabilitate

Fațadele se folosesc atunci când se dorește furnizarea unei interfețe simple pentru un sistem complex. Pe măsură ce evoluează, sistemele devin de obicei din ce în ce mai complexe. Atunci când sunt folosite, majoritatea modelelor sunt completate cu numeroase clase suplimentare, ceea ce face sistemul mai uşor de personalizat, dar de asemenea devine tot mai greu de utilizat pentru clienții care nu au nevoie să îl personalizeze. O fațadă poate furniza o interfață implicită simplă a subsistemului, suficient de bună pentru majoritatea clienților. Doar clienții care au nevoie de mai multă flexibilitate trebuie să utilizeze direct clasele subsistemului din spatele fațadei.

De asemenea, un astfel de subsistem conține de multe ori interdependențe complexe. Folosirea sa ca atare ar creşte cuplarea dintre codul clientului şi elementele subsistemului. Fațada decuplează subsistemul de clienți, deoarece în acest caz clienții depind numai de fațadă.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Utilizarea şablonului poate ajuta şi structurarea pe niveluri a subsistemelor, folosind câte o faţadă pentru a defini câte un punct de intrare pentru fiecare nivel al subsistemului. Dacă nivelurile subsistemului comunică doar prin faţade, cuplarea dintre ele va scădea de asemenea.

4. Analiza şablonului

Structura şablonului *Faţadă* este prezentată în figura 9.2.

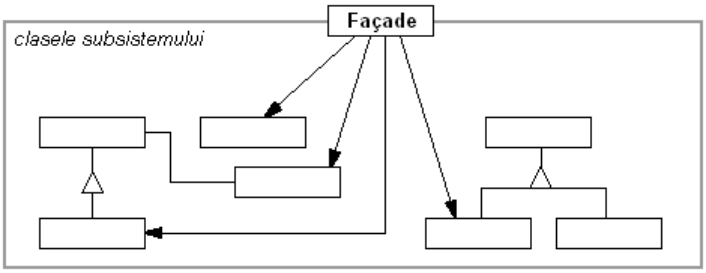


Figura 9.2. *Structura şablonului Faţadă*

Considerând din nou elementele subsistemului compilator, pentru exemplificare, participanţii şablonului sunt:

- *Faţada* (Compiler)
  - Ştie ce funcţionalităţi implementează clasele subsistemului;
  - Trimite cererile clientului către obiectele subsistemului;
- Clasele subsistemului (Scanner, Parser, ProgrameNode etc.)
  - Implementează funcţionalitatea subsistemului;
  - Primesc apeluri de la obiectul *Faţadă*;
  - Nu au cunoştinţă despre *Faţadă*; apelurile sunt unidirecţionale: doar dinspre *Faţadă* către subsistem.

Clienţii comunică cu subsistemul trimiţând cereri faţadei, care le trimite mai departe către obiectele subsistemului ce îndeplinesc efectiv sarcinile. Faţada trebuie să gestioneze interfaţa cu clientul şi să cunoască interfeţele subsistemului.

Clienţii care utilizează faţada nu au nevoie să acceseze obiectele subsistemului, însă acestea nu sunt ascunse. Rămâne la latitudinea clientului dacă doreşte să utilizeze interfaţa simplificată a faţadei sau să apeleze direct funcţionalităţile de nivel scăzut ale obiectelor subsistemului.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

5. Exemplu de implementare

Codul C# corespunzător structurii şablonului *Fațadă* este prezentat mai jos.

Subsistemele

```
class SubsystemOne
{
    public void MethodOne()
    {
        Console.WriteLine("Metoda subsistemului 1 ");
    }
}

class SubsystemTwo
{
    public void MethodTwo()
    {
        Console.WriteLine("Metoda subsistemului 2");
    }
}

class SubsystemThree
{
    public void MethodThree()
    {
        Console.WriteLine("Metoda subsistemului 3");
    }
}

class SubsystemFour
{
    public void MethodFour()
    {
        Console.WriteLine("Metoda subsistemului 4");
    }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Faţada

```
class Facade
{
    SubSystemOne _one;
    SubSystemTwo _two;
    SubSystemThree _three;
    SubSystemFour _four;

    public Facade()
    {
        _one = new SubSystemOne();
        _two = new SubSystemTwo();
        _three = new SubSystemThree();
        _four = new SubSystemFour();
    }

    public void MethodA()
    {
        Console.WriteLine("Metoda A a Fatadei");
        _one.MethodOne();
        _two.MethodTwo();
        _four.MethodFour();
    }

    public void MethodB()
    {
        Console.WriteLine("Metoda B a Fatadei");
        _two.MethodTwo();
        _three.MethodThree();
    }
}
```

Clientul

```
public class Client
{
    public static void Main(string[] args)
    {
        Facade f = new Facade();
        f.MethodA();
        f.MethodB();
        Console.ReadLine();
    }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

6. Aplicație

6.1. Să se realizeze simularea unei comenzi de cumpărare online a unor produse. Există clase pentru activități precum înregistrarea cererii, facturare, expediere, statistici privind cumpărătorii. Clasa Facade se constituie într-un punct de intrare unic pentru toate aceste activități.

Indicații. Diagrama de clase a aplicației este prezentată în figura 9.3.

Pentru inițializarea controlului checkedListBox din fereastra principală se pot utiliza în constructor numele produselor furnizate de clasa InfoProduse. Deoarece dorim să decuplăm clientul (MainForm) de clasele sistemului, numele sunt primite tot prin intermediul *Fațadei*:

```
public MainForm()
{
    InitializeComponent();
    string[] produse = Facade.Produse();
    for (int i = 0; i < produse.Length; i++)
        checkedListBoxProduse.Items.Add(produse[i], false);
}
```

În evenimentul de apăsare a butonului se validează (opțional) CNP-ul prin intermediul metodei statice a *Fațadei*, apoi se instanțiază o *Fațadă* cu datele citite din *text-box*-urile ferestrei principale şi se procesează comanda. O comandă nu trebuie facturată şi expediată dacă nu s-a comandat niciun produs.

Pentru statistici, se prelucrează codul numeric personal: SAALLZZJJNNNC, unde prima cifră reprezintă sexul (impar = *M*, par = *F*), următoarele 6 cifre reprezintă data naşterii (an, lună, zi), următoarele 2 cifre reprezintă codul județului, următoarele 3 indică numărul de ordine din registrul stării civile iar ultima e cifra de control. Persoanele născute între 1900-1999 au prima cifră 1 sau 2, iar cele născute după 2000 au prima cifră 5 sau 6.

Validarea CNP-ului se poate face într-o metodă de tipul:

```
private bool CNPValid()
{
   string cnp = textBoxCNP.Text;

   if (cnp.Length != 13)
      return false;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

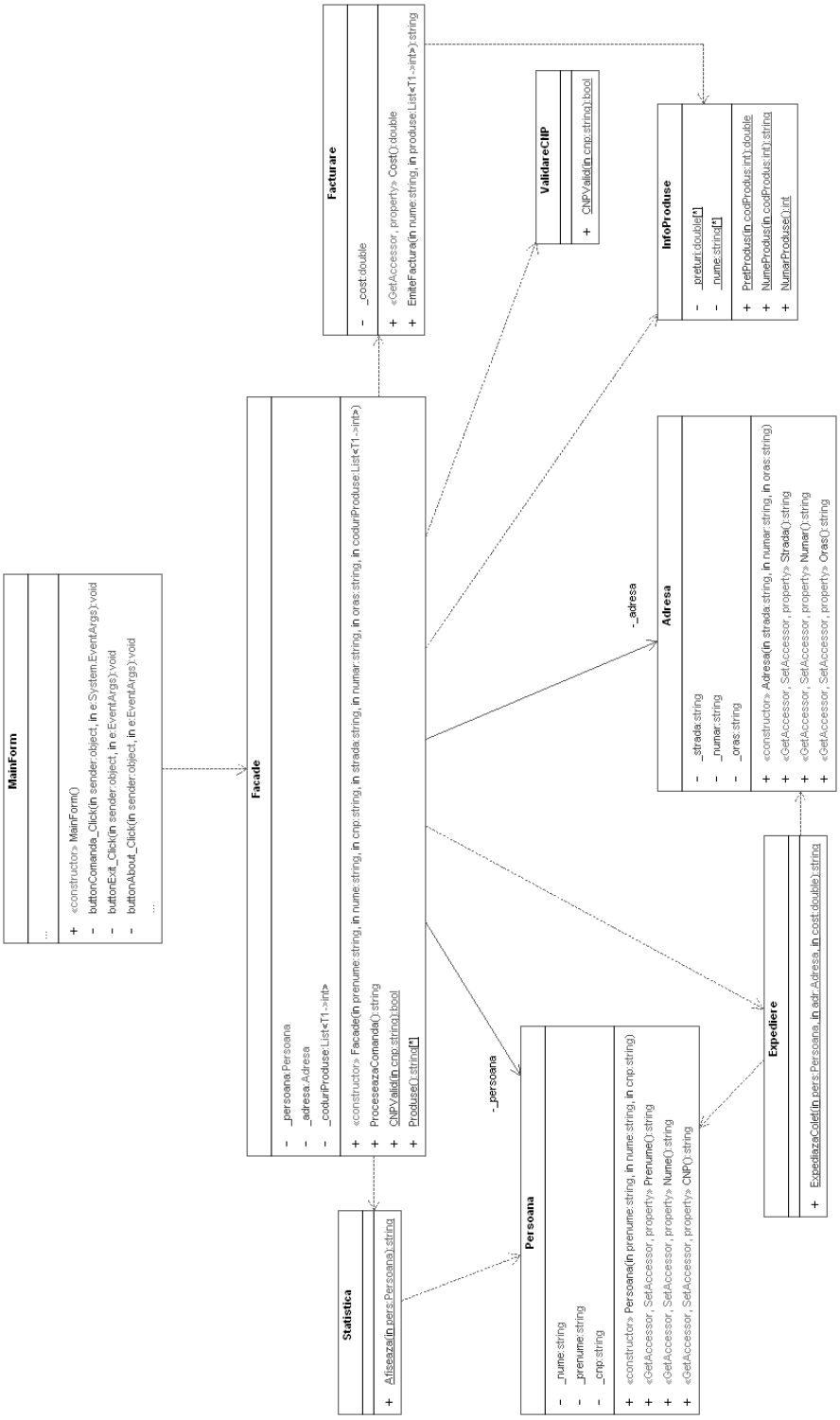


***Figura 9.3.** Exemplu de rezolvare: diagrama de clase*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    int suma = Convert.ToInt32(cnp[0].ToString()) * 2 +
        Convert.ToInt32(cnp[1].ToString()) * 7 +
        Convert.ToInt32(cnp[2].ToString()) * 9 +
        Convert.ToInt32(cnp[3].ToString()) * 1 +
        Convert.ToInt32(cnp[4].ToString()) * 4 +
        Convert.ToInt32(cnp[5].ToString()) * 6 +
        Convert.ToInt32(cnp[6].ToString()) * 3 +
        Convert.ToInt32(cnp[7].ToString()) * 5 +
        Convert.ToInt32(cnp[8].ToString()) * 8 +
        Convert.ToInt32(cnp[9].ToString()) * 2 +
        Convert.ToInt32(cnp[10].ToString()) * 7 +
        Convert.ToInt32(cnp[11].ToString()) * 9;

    int rest = suma % 11;

    if ((rest < 10) && (rest.ToString() == cnp[12].ToString()) ||
        (rest == 10) && (cnp[12] == '1'))
        return true;

    return false;
}
```

Interfața cu utilizatorul poate arăta ca în figura 9.4.

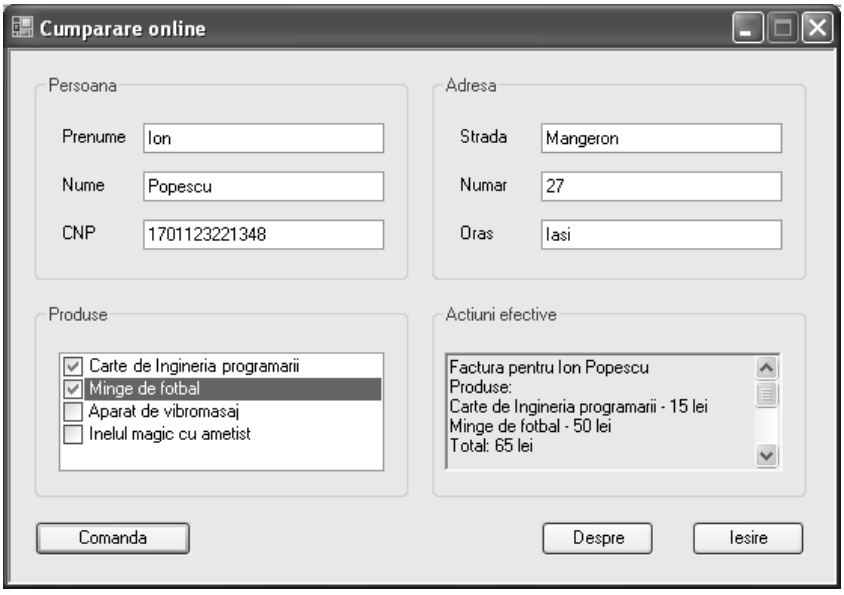


Figura 9.4. *Exemplu de rezolvare: diagrama de clase*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 10

Şablonul de proiectare Proxy

1. Obiectiv
2. Scop şi motivaţie
3. Aplicabilitate
4. Analiza şablonului
5. Exemplu de implementare
6. Aplicaţii

1. Obiectiv

Obiectivul capitolului 10 este implementarea unui program după şablonul de proiectare structural *Proxy* (o traducere posibilă ar fi „Intermediar" sau „Delegare"). Vom considera varianta *proxy*-ului de protecţie pentru stabilirea drepturilor de acces la nişte documente protejate. Tema pentru acasă completează şablonul cu varianta *proxy*-ului la distanţă, pentru accesarea distribuită a documentelor.

2. Scop şi motivaţie

Şablonul *Proxy* asigură un înlocuitor pentru alt obiect pentru a controla accesul la acesta.

Unul din motivele pentru controlarea accesului la un obiect este întârzierea creării şi iniţializării lui până în momentul în care acesta trebuie utilizat. De exemplu, să considerăm un program de editare de documente care poate conţine obiecte grafice. Crearea unor obiecte-imagine raster de mari dimensiuni poate fi foarte costisitoare din punct de vedere al timpului necesar citirii fişierelor de pe harddisk, mai ales dacă presupunem că sunt stocate în format *bmp*. Deschiderea unui document trebuie să fie însă o operaţie rapidă, deci crearea tuturor acestor obiecte costisitoare odată cu deschiderea documentului trebuie evitată. Acest lucru nici nu este necesar, deoarece nu toate imaginile vor fi vizibile în acelaşi timp.

O soluţie este crearea *la cerere* a fiecărui obiect costisitor, în cazul nostru acest lucru având loc în momentul în care o imagine devine vizibilă.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Se pune problema ce vom plasa în document în locul imaginii reale şi cum putem ascunde faptul că imaginea este creată la cerere, astfel încât să nu complicăm implementarea editorului iar optimizarea să nu afecteze codul de formatare şi redare a imaginilor.

Se poate utiliza un alt obiect, un *proxy* de imagine, care să acționeze ca înlocuitor al imaginii reale. Obiectul *proxy* acționează la fel ca imaginea şi o instanțiază când este nevoie, adică doar atunci când programul de editare îi cere să o afişeze. Cererile ulterioare sunt transmise direct către imaginea reală şi deci *proxy*-ul trebuie să aibă o referință către aceasta (figura 10.1).

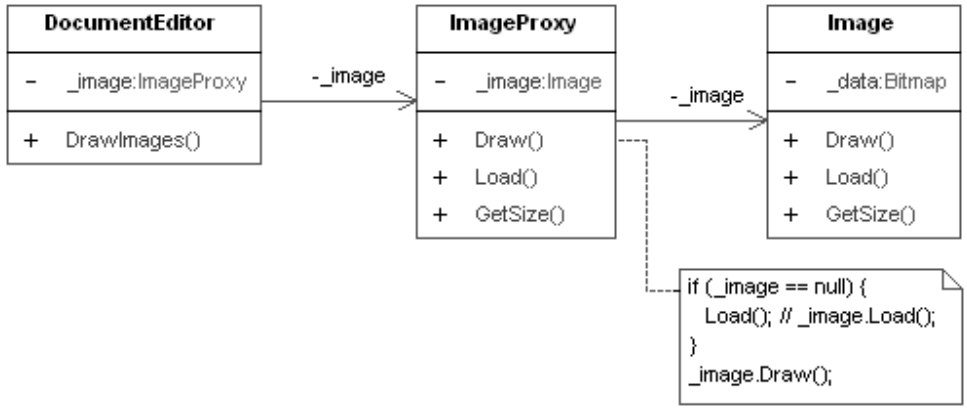


Figura 10.1. *Diagrama de clase a unui proxy de imagine*

Referința la obiectul _image din ImageProxy va fi nulă până în momentul în care editorul apelează metoda Draw şi obiectul *proxy* instanțiază imaginea reală. Operația Draw asigură faptul că imaginea este instanțiată înainte de a fi apelat obiectul Image.

Până atunci, dacă mai există o metodă GetSize care returnează dimensiunile imaginii, obiectul ImageProxy poate returna o valoare proprie implicită. După instanțierea imaginii raster, obiectul ImageProxy va returna dimensiunile reale ale imaginii apelând metoda GetSize din Image.

3. Aplicabilitate

Şablonul *Proxy* poate fi aplicat oriunde este nevoie de o referință mai sofisticată la un obiect. Sunt prezentate în continuare câteva situații în care se poate aplica acest şablon:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

- ***Proxy*-ul la distanţă** (engl. "remote proxy") reprezintă un obiect local care intermediază accesul la un obiect de pe o altă maşină. Este util în general pentru aplicațiile distribuite;
- ***Proxy*-ul virtual** (engl. "virtual proxy") creează la cerere obiecte costisitoare. Un exemplu este clasa ImageProxy din secțiunea anterioară. Imaginea reală poate fi însă şi o imagine descărcată de pe internet; în acest caz *proxy*-ul poate afişa o imagine temporară până când este descărcată imaginea reală. Şablonul nu se referă doar la imagini, ci la orice obiect a cărui creare necesită timp sau resurse importante;
- ***Proxy*-ul de protecţie** (engl. "protection proxy") controlează accesul la obiectul real. Este util atunci când obiectul reprezentat presupune drepturi diferite de acces;
- **Referinţa inteligentă** (engl. "smart reference") este un înlocuitor pentru o referință, care efectuează acțiuni suplimentare în momentul accesării unui obiect. De exemplu, se poate folosi pentru păstrarea cópiilor unor obiecte mari care pot sau nu să se modifice. Dacă se doreşte crearea unei alte instanțe de acest tip, *proxy*-ul poate decide să nu facă efectiv copia, folosind în continuare primul obiect. Dacă clientul încearcă să facă modificări în al doilea obiect, *proxy*-ul face abia atunci copia şi modificările. De asemenea, se poate stabili astfel numărul de referințe către obiectul real, astfel încât acesta să poată fi automat eliberat atunci când nu mai există referințe.

4. Analiza şablonului

Diagrama generică de clase a şablonului *Proxy* este prezentată în figura 10.2.

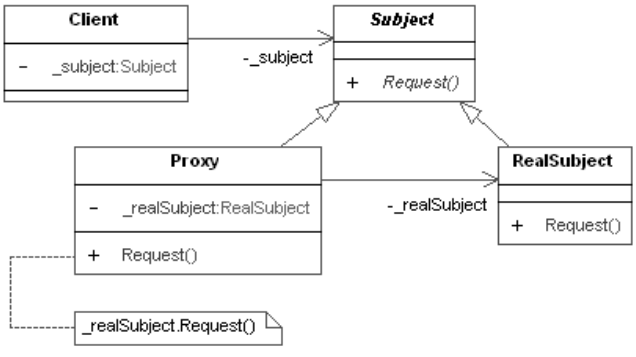


Figura 10.2. *Diagrama de clase a şablonului Proxy*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Trebuie precizat că atât clasa Proxy cât şi clasa RealSubject au o interfață identică, definită de clasa Subject, astfel încât un obiect *proxy* să poată fi înlocuit de obiectul „real”. În funcție de tipul său, obiectul Proxy trimite apelurile către obiectul RealSubject.

5. Exemplu de implementare

Codul C# corespunzător structurii şablonului *Proxy* este prezentat mai jos.

Subiectul abstract

```
abstract class Subject
{
   // Metode
   abstract public void Request();
}
```

Subiectul real

```
class RealSubject : Subject
{
   // Metode
   override public void Request()
   {
      Console.WriteLine("Apelul din subiectul real");
   }
}
```

Proxy

```
class Proxy : Subject
{
   RealSubject _realSubject;

   override public void Request()
   {
      // Se foloseşte iniţializarea întârziată
      if (_realSubject == null)
         _realSubject = new RealSubject();
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
      _realSubject.Request();
   }
}
```

Clientul

```
public class Client
{
   public static void Main(string[] args)
   {
      // Se creează un proxy şi se apelează metoda dorită
      Proxy p = new Proxy();
      p.Request();
   }
}
```

6. Aplicaţii

6.1. Realizați un program pentru accesul la o serie de documente potrivit nivelului de acces al utilizatorului. Un schelet al aplicației pentru construirea interfaței grafice cu utilizatorul (prezentată în figura 10.3) şi fişierele de configurare sunt incluse după descrierea cerințelor şi indicațiilor.

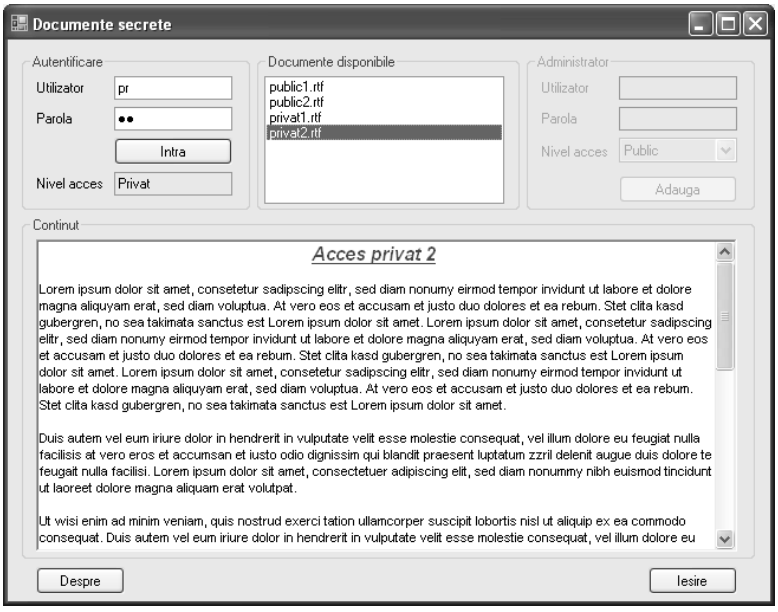


Figura 10.3. *Exemplu de rezolvare: interfaţa cu utilizatorul*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Structură propusă de fişiere este prezentată în figura 10.4.

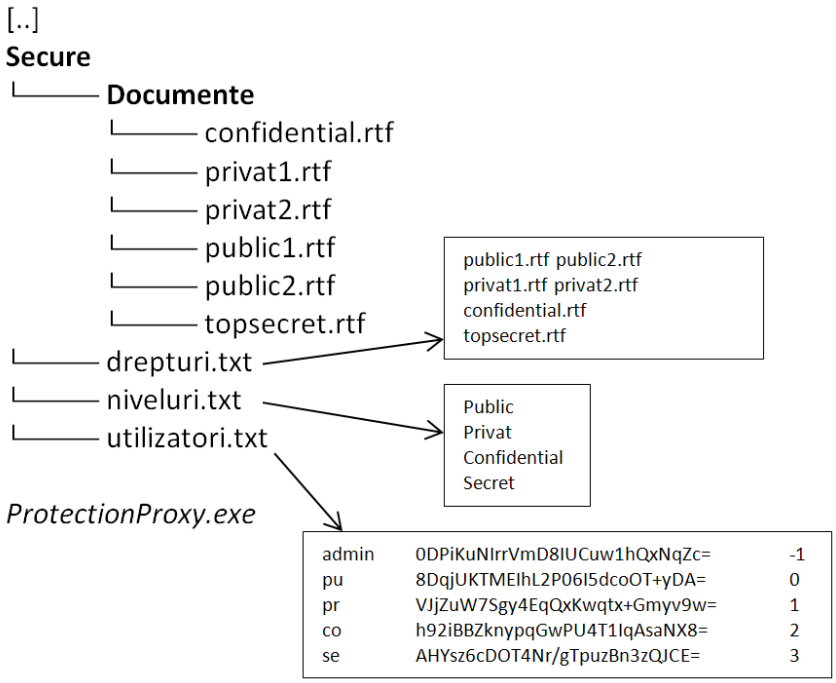


Figura 10.4. *Structura de fişiere a aplicaţiei*

Nivelurile de acces se dau în fişierul *niveluri.txt*:

```
Public
Privat
Confidential
Secret
```

Lista de utilizatori se găseşte în fişierul *utilizatori.txt*, care este de forma specificată în tabelul 10.1.

Tabelul 10.1. *Exemplu de listă de utilizatori*

Numele utilizatorului	Hash-ul parolei	Nivelul de acces
admin	0DPiKuNIrrVmD8IUCuw1hQxNqZc=	-1
pu	8DqjUKTMEIhL2P06I5dcoOT+yDA=	0
pr	VJjZuW7Sgy4EqQxKwqtx+Gmyv9w=	1
se	AHYsz6cDOT4Nr/gTpuzBn3zQJCE=	3
co	h92iBBZknypqGwPU4T1IqAsaNX8=	2

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Pe prima coloană este numele utilizatorului. Pe coloana a doua este *hash*-ul parolei, pentru evitarea stocării în clar a acesteia. La autentificare, se va calcula *hash*-ul parolei şi se va compara cu *hash*-ul din fişier. Pe a treia coloană se notează nivelul de acces. Administratorul este un tip special de utilizator. Acesta nu poate consulta documente, dar poate adăuga utilizatori. Inițial, numele administratorului este *admin* şi parola este tot *admin*. În situația de mai sus, parolele utilizatorilor sunt egale cu numele lor. În program, la autentificarea unui utilizator se verifică existența utilizatorului şi corectitudinea parolei.

Documentele din directorul *Documente* au asociate şi ele drepturi de acces, setate în fişierul *drepturi.txt*:

```
public1.rtf public2.rtf
privat1.rtf privat2.rtf
confidential.rtf
topsecret.rtf
```

Pe prima linie sunt documentele corespunzătoare primului nivelul de acces (*Public*), pe linia a doua documentele pentru nivelul *Privat* ş.a.m.d. Când un utilizator se autentifică, el primeşte lista documentelor de pe nivelul său de acces dar şi documentele de pe nivelurile inferioare. Se presupune că serverul de documente ar fi pe altă maşină. La selectarea unui document, acesta este criptat şi trimis clientului care îl decriptează şi îl afişează. Accesul la documente se va face printr-un *proxy* de protecție. Acesta la rândul său accesează managerul real de documente. Clasele ProxyDocumentManager şi RealDocumentManager vor implementa ambele interfața IDocumentManager.

Diagrama de clase a aplicației este prezentată în figura 10.5.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

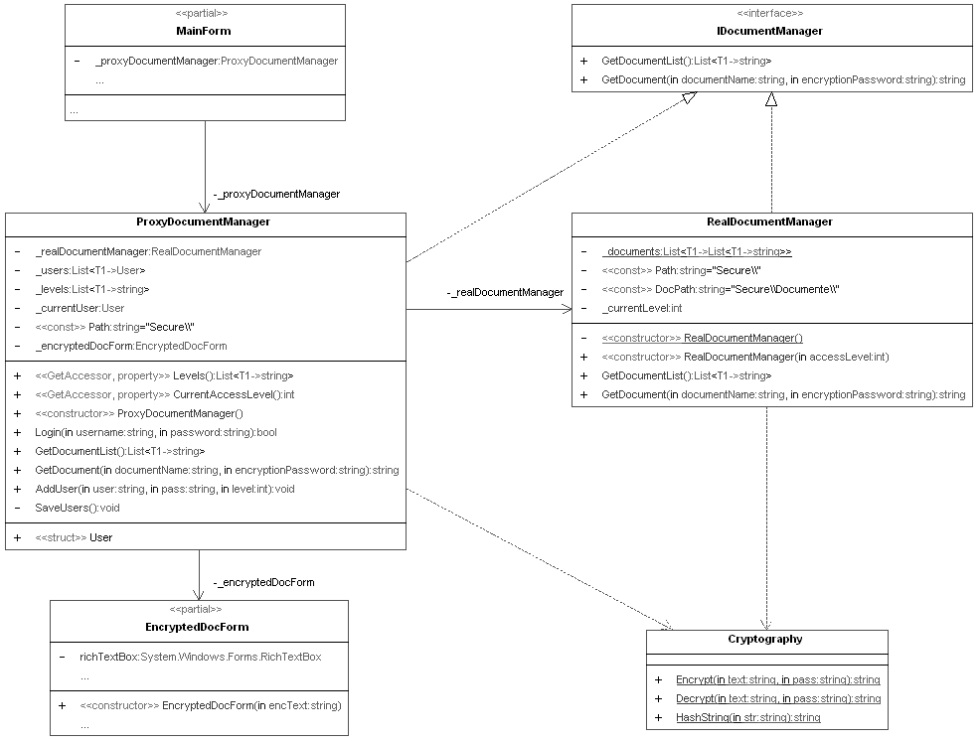


Figura 10.5. *Exemplu de rezolvare: diagrama de clase*

ProxyDocumentManager se ocupă de autentificare şi gestionarea drepturilor de acces. Metodele GetDocumentList şi GetDocument apelează metodele corespunzătoare din RealDocumentManager. RealDocumentManager criptează documentul şi îl trimite către ProxyDocumentManager. Această operaţie este importantă mai ales atunci când ProxyDocumentManager şi RealDocumentManager se află pe maşini diferite (situaţia de la temă).

Atunci când ProxyDocumentManager primeşte documentul, îi afişează conţinutul criptat în EncryptedDocForm. Este o operaţie opţională, utilă doar pentru a vedea mai clar aspectul unui fişier binar transformat ca *string* în „baza 64”, prin apelul metodei ToBase64String. Pentru a nu incomoda, această fereastră secundară poate fi implicit minimizată:

```
_encryptedDocForm = new EncryptedDocForm(encryptedFile);
_encryptedDocForm.WindowState = FormWindowState.Minimized;
_encryptedDocForm.Show();
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

În vederea afişării documentului decriptat într-un control de tip RichTextBox de către MainForm, se utilizează proprietatea Rtf a controlului, de exemplu: richTextBox.Rtf = decryptedRtfDoc.

Pentru a vă putea concentra exclusiv asupra aplicării şablonului, în continuare se dau câteva clase ce urmează a fi completate sau utilizate în program.

MainForm.cs

```
namespace ProtectionProxy
{
   public partial class MainForm : Form
   {
      private ProxyDocumentManager _proxyDocumentManager;

      public MainForm()
      {
         InitializeComponent();
         groupBoxAdmin.Enabled = false;
         _proxyDocumentManager = new ProxyDocumentManager();
      }
   }
}
```

DocumentManager.cs

```
namespace ProtectionProxy
{
   interface IDocumentManager
   {
      List<string> GetDocumentList();
      string GetDocument(string documentName, string encryptionPassword);
   }
}
```

ProxyDocumentManager.cs

```
namespace ProtectionProxy
{
   public class ProxyDocumentManager : IDocumentManager
   {
      private RealDocumentManager _realDocumentManager;
      private List<User> _users;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
private List<string> _levels;
private const string Path = "Secure\\";

public struct User
{
   public readonly string Name;
   public readonly string PassHash;
   public readonly int AccessLevel;

   public User(string name, string passHash, int accessLevel)
   {
      Name = name;
      PassHash = passHash;
      AccessLevel = accessLevel;
   }
}

public ProxyDocumentManager()
{
   _levels = new List<string>();
   StreamReader sr = new StreamReader(Path + "niveluri.txt");
   string[] lvls = sr.ReadToEnd().Split(" \t\r\n".ToCharArray(),
      StringSplitOptions.RemoveEmptyEntries);
   sr.Close();

   for (int i = 0; i < lvls.Length; i++)
      _levels.Add(lvls[i]);

   _users = new List<User>();
   sr = new StreamReader(Path + "utilizatori.txt");
   string line;
   while ((line = sr.ReadLine()) != null)
   {
      string[] toks = line.Split('\t');
      User user = new User(toks[0], toks[1], Convert.ToInt32(toks[2]));
      _users.Add(user);
   }
   sr.Close();
}

#region IDocumentManager Members

public List<string> GetDocumentList()
{
   throw new Exception("The method or operation is not implemented.");
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
        public string GetDocument(string documentName, string encryptionPassword)
        {
            throw new Exception("The method or operation is not implemented.");
        }

        #endregion
    }
}
```

RealDocumentManager.cs

```
namespace ProtectionProxy
{
    class RealDocumentManager : IDocumentManager
    {
        private static List<List<string>> _documents;
        private const string Path = "Secure\\", DocPath = "Secure\\Documente\\";

        static RealDocumentManager()
        {
            int numberOfLevels = 0;
            StreamReader sr = new StreamReader(Path + "drepturi.txt");
            string[] lines = sr.ReadToEnd().Split("\r\n".ToCharArray(),
                StringSplitOptions.RemoveEmptyEntries);
            sr.Close();
            numberOfLevels = lines.Length;

            _documents = new List<List<string>>(numberOfLevels);
            for (int i = 0; i < numberOfLevels; i++)
                _documents.Add(new List<string>());

            sr = new StreamReader(Path + "drepturi.txt");
            for (int i = 0; i < numberOfLevels; i++)
            {
                string[] files = sr.ReadLine().Split();
                for (int j = i; j < numberOfLevels; j++)
                {
                    for (int k = 0; k < files.Length; k++)
                        _documents[j].Add(files[k]);
                }
            }
            sr.Close();
        }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    #region IDocumentManager Members
    public List<string> GetDocumentList()
    {
      throw new Exception("The method or operation is not implemented.");
    }

    public string GetDocument(string documentName, string encryptionPassword)
    {
      throw new Exception("The method or operation is not implemented.");
    }

    #endregion
  }
}
```

Observaţii: parola pentru criptarea unui documentului transmis se poate considera cunoscută de către ProxyDocumentManager şi RealDocumentManager, astfel încât să nu mai apară ca parametru în metoda GetDocument (encryptionPassword). Stabilirea în mod dinamic a unei parole comune între două entităţi care comunică printr-un canal public este o problemă în sine şi nu poate fi tratată aici. De asemenea, parola folosită pentru criptarea documentelor este diferită de parola utilizatorului şi nici nu trebuie să fie cunoscută de către acesta.

Constructorul static al clasei RealDocumentManager asigură faptul că listele cu drepturile de acces vor fi iniţializate o singură dată înainte de utilizarea efectivă a clasei de către ProxyDocumentManager. Constructorul static este apelat automat înainte de a fi creată prima instanţă a clasei sau înainte de referenţierea membrilor statici.

Cryptography.cs

```
/*********************************************************************
 *  Website:     http://www.obviex.com/samples/Encryption.aspx          *
 *               Adaptation and added functionality by Florin Leon      *
 *               http://florinleon.byethost24.com/lab_ip.htm            *
 *  Description: Contains functions for encryption, decryption,         *
 *               and hashing.                                           *
*********************************************************************/

using System;
using System.Security.Cryptography;
using System.IO;
using System.Text;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
namespace ProtectionProxy
{
   public class Cryptography
   {
      /// <summary>
      /// Encrypts specified plaintext using Rijndael symmetric key algorithm
      /// and returns a base64-encoded result.
      /// </summary>
      /// <param name="plainText">
      /// Plaintext value to be encrypted.
      /// </param>
      /// <param name="passPhrase">
      /// Passphrase from which a pseudo-random password will be derived. The
      /// derived password will be used to generate the encryption key.
      /// Passphrase can be any string. In this example we assume that this
      /// passphrase is an ASCII string.
      /// </param>
      /// <param name="saltValue">
      /// Salt value used along with passphrase to generate password. Salt can
      /// be any string. In this example we assume that salt is an ASCII string.
      /// </param>
      /// <param name="hashAlgorithm">
      /// Hash algorithm used to generate password. Allowed values are: "MD5" and
      /// "SHA1". SHA1 hashes are a bit slower, but more secure than MD5 hashes.
      /// </param>
      /// <param name="passwordIterations">
      /// Number of iterations used to generate password. One or two iterations
      /// should be enough.
      /// </param>
      /// <param name="initVector">
      /// Initialization vector (or IV). This value is required to encrypt the
      /// first block of plaintext data. For RijndaelManaged class IV must be
      /// exactly 16 ASCII characters long.
      /// </param>
      /// <param name="keySize">
      /// Size of encryption key in bits. Allowed values are: 128, 192, and 256.
      /// Longer keys are more secure than shorter keys.
      /// </param>
      /// <returns>
      /// Encrypted value formatted as a base64-encoded string.
      /// </returns>
      public static string Encrypt(string text, string pass)
      {
         string plainText = text;
         string passPhrase = pass;
         string saltValue = "s@1tValue"; // can be any string
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
string hashAlgorithm = "SHA1";// can be "MD5"
int passwordIterations = 2; // can be any number
string initVector = "@1B2c3D4e5F6g7H8"; // must be 16 bytes
int keySize = 256; // can be 192 or 128

// Convert strings into byte arrays.
// Let us assume that strings only contain ASCII codes.
// If strings include Unicode characters, use Unicode, UTF7, or UTF8
// encoding.
byte[] initVectorBytes = Encoding.ASCII.GetBytes(initVector);
byte[] saltValueBytes = Encoding.ASCII.GetBytes(saltValue);

// Convert our plaintext into a byte array.
// Let us assume that plaintext contains UTF8-encoded characters.
byte[] plainTextBytes = Encoding.UTF8.GetBytes(plainText);

// First, we must create a password, from which the key will be derived.
// This password will be generated from the specified passphrase and
// salt value. The password will be created using the specified hash
// algorithm. Password creation can be done in several iterations.
PasswordDeriveBytes password = new PasswordDeriveBytes(
                                passPhrase,
                                saltValueBytes,
                                hashAlgorithm,
                                passwordIterations);

// Use the password to generate pseudo-random bytes for the encryption
// key. Specify the size of the key in bytes (instead of bits).
byte[] keyBytes = password.GetBytes(keySize / 8);

// Create uninitialized Rijndael encryption object.
RijndaelManaged symmetricKey = new RijndaelManaged();

// It is reasonable to set encryption mode to Cipher Block Chaining
// (CBC). Use default options for other symmetric key parameters.
symmetricKey.Mode = CipherMode.CBC;

// Generate encryptor from the existing key bytes and initialization
// vector. Key size will be defined based on the number of the key
// bytes.
ICryptoTransform encryptor = symmetricKey.CreateEncryptor(
                                keyBytes,
                                initVectorBytes);

// Define memory stream which will be used to hold encrypted data.
MemoryStream memoryStream = new MemoryStream();
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    // Define cryptographic stream (always use Write mode for encryption).
    CryptoStream cryptoStream = new CryptoStream(memoryStream,
                                    encryptor,
                                    CryptoStreamMode.Write);
    // Start encrypting.
    cryptoStream.Write(plainTextBytes, 0, plainTextBytes.Length);

    // Finish encrypting.
    cryptoStream.FlushFinalBlock();

    // Convert our encrypted data from a memory stream into a byte array.
    byte[] cipherTextBytes = memoryStream.ToArray();

    // Close both streams.
    memoryStream.Close();
    cryptoStream.Close();

    // Convert encrypted data into a base64-encoded string.
    string cipherText = Convert.ToBase64String(cipherTextBytes);

    // Return encrypted string.
    return cipherText;
  }

  /// <summary>
  /// Decrypts specified ciphertext using Rijndael symmetric key algorithm.
  /// </summary>
  /// <param name="cipherText">
  /// Base64-formatted ciphertext value.
  /// </param>
  /// <param name="passPhrase">
  /// Passphrase from which a pseudo-random password will be derived. The
  /// derived password will be used to generate the encryption key.
  /// Passphrase can be any string. In this example we assume that this
  /// passphrase is an ASCII string.
  /// </param>
  /// <param name="saltValue">
  /// Salt value used along with passphrase to generate password. Salt can
  /// be any string. In this example we assume that salt is an ASCII string.
  /// </param>
  /// <param name="hashAlgorithm">
  /// Hash algorithm used to generate password. Allowed values are: "MD5" and
  /// "SHA1". SHA1 hashes are a bit slower, but more secure than MD5 hashes.
  /// </param>
  /// <param name="passwordIterations">
  /// Number of iterations used to generate password. One or two iterations
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    /// should be enough.
    /// </param>
    /// <param name="initVector">
    /// Initialization vector (or IV). This value is required to encrypt the
    /// first block of plaintext data. For RijndaelManaged class IV must be
    /// exactly 16 ASCII characters long.
    /// </param>
    /// <param name="keySize">
    /// Size of encryption key in bits. Allowed values are: 128, 192, and 256.
    /// Longer keys are more secure than shorter keys.
    /// </param>
    /// <returns>
    /// Decrypted string value.
    /// </returns>
    /// <remarks>
    /// Most of the logic in this function is similar to the Encrypt
    /// logic. In order for decryption to work, all parameters of this function
    /// - except cipherText value - must match the corresponding parameters of
    /// the Encrypt function which was called to generate the
    /// ciphertext.
    /// </remarks>
    public static string Decrypt(string text, string pass)
    {
        string cipherText = text;
        string passPhrase = pass;
        string saltValue = "s@1tValue"; // can be any string
        string hashAlgorithm = "SHA1";// can be "MD5"
        int passwordIterations = 2; // can be any number
        string initVector = "@1B2c3D4e5F6g7H8"; // must be 16 bytes
        int keySize = 256; // can be 192 or 128
        // Convert strings defining encryption key characteristics into byte
        // arrays. Let us assume that strings only contain ASCII codes.
        // If strings include Unicode characters, use Unicode, UTF7, or UTF8
        // encoding.
        byte[] initVectorBytes = Encoding.ASCII.GetBytes(initVector);
        byte[] saltValueBytes = Encoding.ASCII.GetBytes(saltValue);

        // Convert our ciphertext into a byte array.
        byte[] cipherTextBytes = Convert.FromBase64String(cipherText);

        // First, we must create a password, from which the key will be
        // derived. This password will be generated from the specified
        // passphrase and salt value. The password will be created using
        // the specified hash algorithm. Password creation can be done in
        // several iterations.
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
PasswordDeriveBytes password = new PasswordDeriveBytes(
                                passPhrase,
                                saltValueBytes,
                                hashAlgorithm,
                                passwordIterations);

// Use the password to generate pseudo-random bytes for the encryption
// key. Specify the size of the key in bytes (instead of bits).
byte[] keyBytes = password.GetBytes(keySize / 8);

// Create uninitialized Rijndael encryption object.
RijndaelManaged symmetricKey = new RijndaelManaged();

// It is reasonable to set encryption mode to Cipher Block Chaining
// (CBC). Use default options for other symmetric key parameters.
symmetricKey.Mode = CipherMode.CBC;

// Generate decryptor from the existing key bytes and initialization
// vector. Key size will be defined based on the number of the key
// bytes.
ICryptoTransform decryptor = symmetricKey.CreateDecryptor(
                                 keyBytes,
                                 initVectorBytes);

// Define memory stream which will be used to hold encrypted data.
MemoryStream memoryStream = new MemoryStream(cipherTextBytes);

// Define cryptographic stream (always use Read mode for encryption).
CryptoStream cryptoStream = new CryptoStream(memoryStream,
                                decryptor,
                                CryptoStreamMode.Read);

// Since at this point we don't know what the size of decrypted data
// will be, allocate the buffer long enough to hold ciphertext;
// plaintext is never longer than ciphertext.
byte[] plainTextBytes = new byte[cipherTextBytes.Length];

// Start decrypting.
int decryptedByteCount = cryptoStream.Read(plainTextBytes,
                             0,
                             plainTextBytes.Length);

// Close both streams.
memoryStream.Close();
cryptoStream.Close();
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
        // Convert decrypted data into a string.
        // Let us assume that the original plaintext string was UTF8-encoded.
        string plainText = Encoding.UTF8.GetString(plainTextBytes,
                                                   0,
                                                   decryptedByteCount);

        // Return decrypted string.
        return plainText;
      }

      /// <summary>
      /// Returns the hash of a string
      /// </summary>
      /// <param name="str"></param>
      /// <returns></returns>
      public static string HashString(string str)
      {
        HashAlgorithm sha = new SHA1CryptoServiceProvider();
        byte[] buf = new byte[str.Length];
        for (int i = 0; i < str.Length; i++)
          buf[i] = (byte)str[i];
        byte[] result = sha.ComputeHash(buf);
        return Convert.ToBase64String(result);
      }
    }
}
```

6.2. *Temă pentru acasă*. Extindeți programul astfel încât documentele şi managerul de documente să poată fi pe un alt calculator iar *proxy*-ul să înglobeze şi funcționalități de delegare la distanță. Opțional, pentru eficientizarea comunicațiilor, se poate avea în vedere arhivarea documentului trimis *înainte* de criptare.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 11

Şablonul de proiectare Comandă

1. Obiective
2. Scop şi motivaţie
3. Aplicabilitate
4. Analiza şablonului
5. Exemplu de implementare
6. Aplicaţie

1. Obiective

Obiectivul principal al capitolului 11 este implementarea unui program după şablonul de proiectare comportamental *Comandă* (engl. "Command"). Se va prezenta şi o metodă de generare şi adăugare dinamică a unor controale în fereastra unui program.

2. Scop şi motivaţie

Şablonul *Comandă* încapsulează o funcţie într-un obiect, astfel încât este posibilă trimiterea unor funcţii ca parametri, formarea de cozi de apeluri, înregistrarea listei de apeluri (engl. "logging") şi asigurarea suportului pentru operaţiile de anulare (engl. "undo").

Să presupunem o aplicaţie cu o bară de instrumente care poate fi personalizată. Cu fiecare buton al barei, utilizatorul îşi poate asocia funcţia dorită. Deci, din punct de vedere al proiectantului, nu se poate cunoaşte apriori nici operaţia propriu-zisă care va fi efectuată de un buton şi nici obiectul concret care va realiza operaţia respectivă. De asemenea, pentru a avea o arhitectură elegantă, toate butoanele trebuie tratate în mod unitar.

Soluţia propusă de şablonul *Comandă* este încapsularea unui astfel de apel într-un obiect, care poate fi stocat sau trimis ca parametru altor obiecte. După cum se poate vedea în figura 11.1, clasa abstractă `Command` include o operaţie `Execute`, implementată de către clasele concrete derivate, care conţin ca şi câmp destinatarul apelului (obiectul care va realiza efectiv operaţia) şi care îi vor transfera acestuia apelul de execuţie a operaţiei. Pentru simplitate, s-au omis semnăturile metodelor. Trebuie subliniat faptul

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

că aceste clase de comandă sunt clase normale, care pot include şi alte câmpuri şi metode pe lângă metoda Execute specifică şablonului.

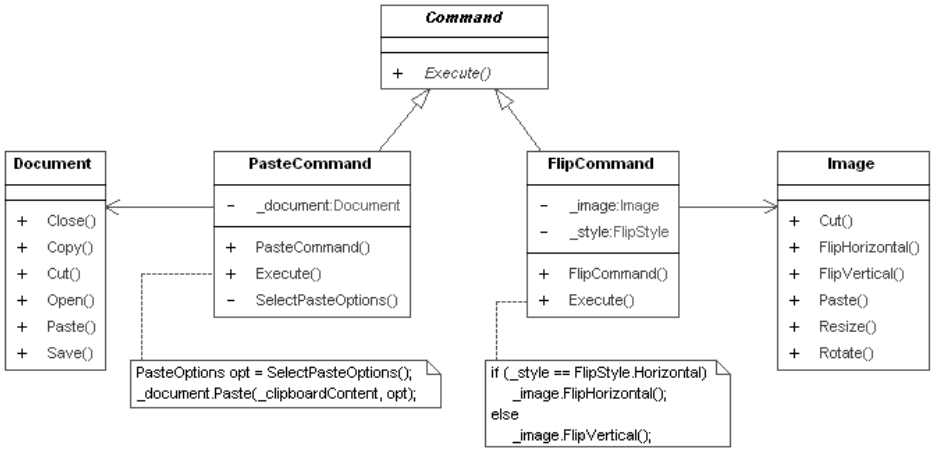


Figura 11.1. *Exemplu de sistem bazat pe şablonul Comandă*

În acest exemplu, destinatarii sunt clasele Document şi Image. Comanda PasteCommand are un câmp de tipul Document iar clasa FlipCommand are un câmp de tipul Image. La apelul metodelor Execute, obiectele de comandă apelează metodele corespunzătoare din obiectele destinatar, după unele eventuale prelucrări suplimentare.

Clasele de comandă sunt derivate dintr-o clasă de bază comună şi deci pot fi tratate unitar, chiar dacă operațiile lor Execute au implementări diferite.

Şablonul permite de asemenea definirea unor comenzi *macro*, adică a unei mulțimi de operații. O astfel de clasă, prezentată în figura 11.2, poate avea un vector de comenzi simple, pe care le apelează succesiv.

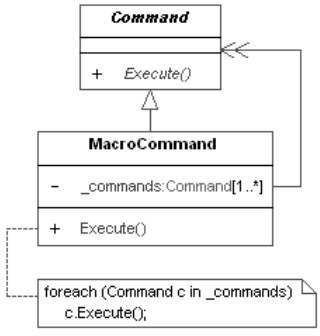


Figura 11.2. *Comandă macro*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Clasa MacroCommand este derivată la rândul său din aceeaşi clasă abstractă Command, deci poate fi tratată ca orice altă comandă. Ea nu are însă un destinatar explicit, deoarece comenzile din serie au propriul destinatar.

În măsura în care sistemul dispune de mai multe comenzi, dacă gestionarea acestora ar aparţine exclusiv clientului, acest fapt ar conduce la creşterea cuplării între aceste clase. De aceea, şablonul introduce o clasă suplimentară, Invoker, care este responsabilă de setarea dinamică a comenzilor şi de apelarea acestora. În exemplul anterior, bara de instrumente este un Invoker, care gestionează comenzile asociate cu butoanele sale, iar aplicaţia (clasa corespunzătoare ferestrei principale, de exemplu) este clientul. Odată ce o comandă a fost introdusă în Invoker, ea poate fi executată (o dată sau de mai multe ori), poate fi eliminată sau înlocuită în mod dinamic.

Una din trăsăturile foarte importante care pot fi implementate uşor cu ajutorul şablonului Comandă este asigurarea operaţiilor de *anulare* (engl. "undo") şi *refacere* (engl. "redo") a ultimelor acţiuni.

În general, chiar dacă aplicaţia are mai multe comenzi, există câte o singură opţiune de anulare, respectiv refacere. De aceea, sistemul trebuie să cunoască succesiunea operaţiilor efectuate şi modul în care starea sistemului este modificată de fiecare comandă. Astfel, clasele de comandă vor avea două metode suplimentare, Undo şi Redo şi vor trebui să reţină starea sistemului înainte de aplicarea comenzii şi după aplicarea sa (figura 11.3).

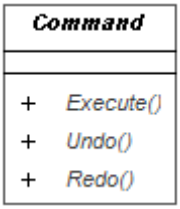


Figura 11.3. *Comandă cu operaţii de anulare şi refacere*

Clasa Invoker poate reţine într-o stivă succesiunea de comenzi efectuate iar dacă se doreşte de exemplu anularea ultimelor *n* acţiuni, se va apela metoda Undo din cele *n* elemente scoase din stivă.

Putem spune că şablonul decuplează obiectul care invocă o operaţie de cel care ştie cum să o efectueze, ceea ce oferă un grad ridicat de flexibilitate în proiectarea unei aplicaţii. În cazul interfeţei grafice cu utilizatorul, o aplicaţie poate furniza pentru o funcţie atât o interfaţă cu meniuri, cât şi una cu butoane şi în acest caz meniurile şi butoanele vor fi

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

asociate cu aceleaşi instanțe ale claselor concrete de comandă. Comenzile pot fi înlocuite în mod dinamic, lucru util pentru implementarea meniurilor sensibile la context. De asemenea, prin compunerea comenzilor într-unele de mai mari dimensiuni, se poate facilita generarea *script*-urilor de comandă.

3. Aplicabilitate

Şablonul *Comandă* poate fi utilizat în primul rând atunci când trebuie să se trimită o acțiune ca parametru unui obiect, aşa cum este cazul în situația cu bara de instrumente prezentată anterior. Trimiterea poate fi făcută şi prin intermediul unei funcții de apelare inversă (engl. "callback"). De exemplu, codul de tratare a evenimentelor în platforma .NET este încapsulat în funcții delegat şi utilizat sub forma:

```
button.Click += new System.EventHandler(button_Click)
```

Comenzile reprezintă un echivalent orientat obiect pentru funcțiile de apelare inversă.

Să presupunem că într-un sistem de echilibrarea încărcării, activitățile de calcul (*task*-urile) trebuie transferate pe diferite maşini. Fiecare activitate de calcul este diferită. Pentru a trata în mod unitar toate aceste activități, ele pot fi încapsulate în obiecte de comandă. La fel, dacă activitățile sunt preluate pentru rulare de fire de execuție dintr-un bazin (engl. "thread pool"), utilizarea comenzilor simplifică foarte mult procesul, deoarece un fir de execuție nu trebuie să ştie decât că o activitate are o metodă `Execute` care trebuie apelată.

După cum am menționat, o comandă poate avea posibilitatea anulării sau refacerii acțiunii. Dacă sistemul trebuie să permită anularea şi refacerea unei întregi succesiuni de comenzi, acest fapt poate fi uşor implementat prin introducerea în `Invoker` a unor liste (sau stive) de comenzi pentru anulare, respectiv refacere. În momentul când se anulează o operație, comanda respectivă se scoate din lista comenzilor de anulare şi se introduce în lista comenzilor de refacere. Se procedează analog în cazul apelării unei operații de refacere. Prin traversarea acestor liste se pot asigura oricâte niveluri de anulare şi refacere.

Pentru sisteme care gestionează date foarte complexe, salvarea stării în cazul blocării se poate face tot cu ajutorul comenzilor. În această situație nu este fezabilă salvarea datelor după fiecare operație efectuată. Aplicația

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

poate să păstreze o înregistrare (engl. "log") cu seria de comenzi efectuate de la pornirea sa. Dacă la un moment dat sistemul se blochează, starea sa poate fi refăcută din cea inițială, aplicând în ordine comenzile salvate pe harddisk. În interfața obiectelor de comandă pot fi adăugate operații de încărcare şi salvare. Salvarea comenzilor poate fi atât o listă cu identificatorii şi parametrii acestora, cât şi serializarea efectivă a obiectelor de comandă aplicate.

4. Analiza şablonului

Diagrama generică de clase a şablonului *Comandă* este prezentată în figura 11.4.

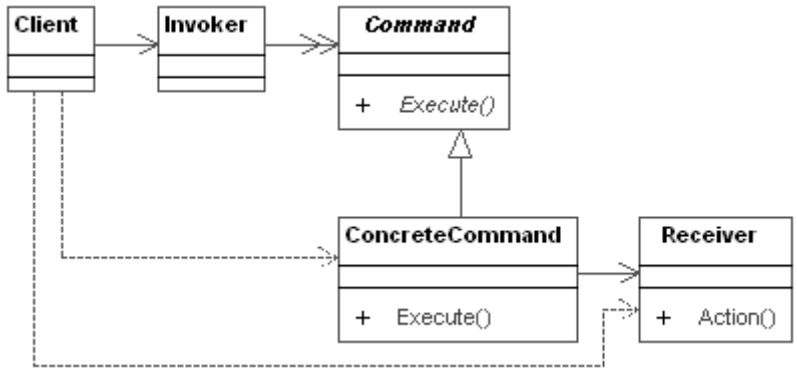


Figura 11.4. *Diagrama de clase a şablonului Comandă*

Clasa abstractă (sau interfața) Command declară metode precum Execute, eventual Undo şi Redo. Clasele concrete ConcreteCommand implementează operația Execute prin invocarea operației sau operațiilor corespunzătoare din obiectul Receiver. Clasa Receiver ştie cum să efectueze operațiile asociate cu executarea unei comenzi. Orice clasă poate servi drept destinatar. Clasa Invoker cere comenzii să execute acțiunea. Clientul creează un obiect ConcreteCommand şi îi stabileşte destinatarul.

Modul în care acționează participanții este detaliat în diagrama de secvențe din figura 11.5.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

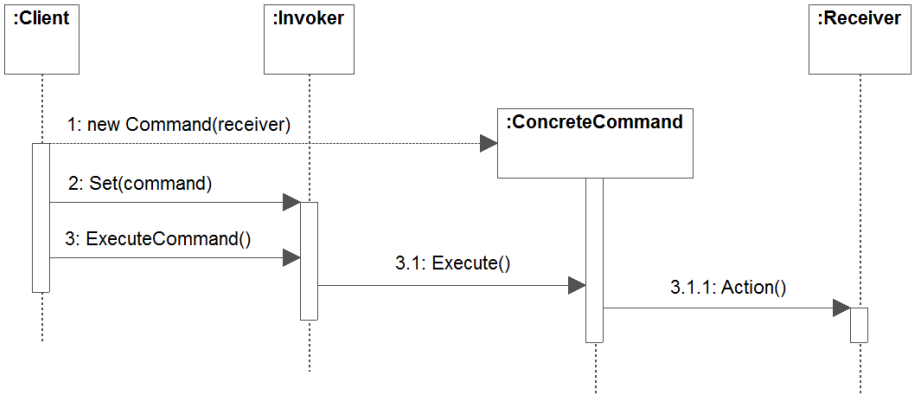


Figure 11.5. *Diagrama de secvențe a șablonului Comandă*

Clientul creează obiectul ConcreteCommand şi îi precizează destinatarul. Obiectul Invoker stochează obiectul ConcreteCommand. Când este nevoie, obiectul Invoker apelează operația Execute din obiectul Command. În cazul în care comenzile pot fi anulate, obiectul ConcreteCommand îşi stochează starea pentru anularea comenzii înainte de a aplica operația Execute. Obiectul ConcreteCommand apelează operațiile din obiectul Receiver pentru a îndeplini acțiunea propriu-zisă.

Consecința principală a şablonului este decuplarea obiectului care invocă o operație de obiectul care ştie cum să o efectueze. Astfel, noi comenzi pot fi adăugate foarte uşor deoarece nu sunt necesare modificări ale claselor existente.

5. Exemplu de implementare

Codul C# corespunzător structurii şablonului *Comandă* este prezentat în cele ce urmează.

Comanda abstractă

```
abstract class Command
{
  protected Receiver _receiver;

  public Command(Receiver receiver)
  {
    _receiver = receiver;
  }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
  abstract public void Execute();
}
```

Comanda concretă

```
class ConcreteCommand : Command
{
  public ConcreteCommand(Receiver receiver) : base(receiver)
  {
  }

  public override void Execute()
  {
    _receiver.Action();
  }
}
```

Destinatarul

```
class Receiver
{
  public void Action()
  {
    Console.WriteLine("Apelul metodei din destinatar");
  }
}
```

Invocatorul

```
class Invoker
{
  private Command _command;

  public void SetCommand(Command command)
  {
    _command = command;
  }

  public void ExecuteCommand()
  {
    _command.Execute();
  }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Clientul

```
public class Client
{
  public static void Main(string[] args)
  {
    // Se creează destinatarul, comanda şi invocatorul
    Receiver r = new Receiver();
    Command c = new ConcreteCommand(r);
    Invoker i = new Invoker();

    // Se setează şi se execută comanda
    i.SetCommand(c);
    i.ExecuteCommand();
  }
}
```

6. Aplicaţii

6.1. Realizaţi un program care simulează o foaie de calcul (gen *Microsoft Excel*). Un schelet al aplicației pentru construirea interfeței grafice cu utilizatorul (prezentată în figura 11.6) este inclus după observaţii şi recomandări.

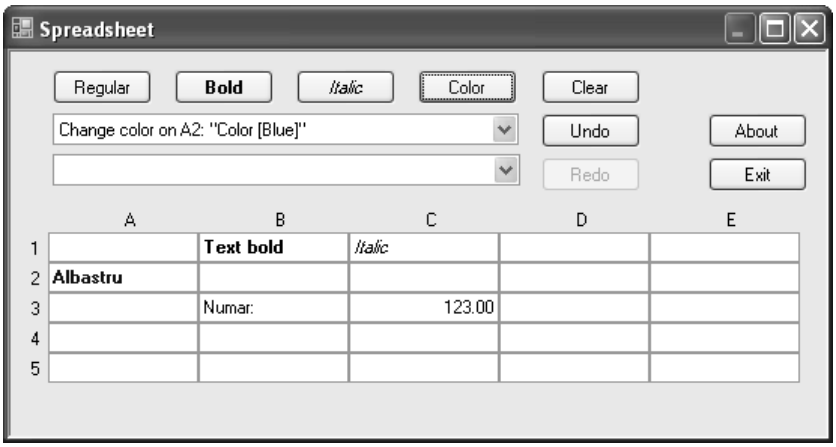


Figura 11.6. *Exemplu de rezolvare: interfaţa cu utilizatorul*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Observaţii:

- Controlul de tip grid este realizat dinamic cu *text-box*-uri. Alternativ, se poate lucra cu un control DataGridView;
- Componentele controlului sunt de fapt de tip ExtendedTextBox, clasă derivată din TextBox, care include o metodă de formatare specială a numerelor din foaia de calcul: cu două zecimale şi aliniate la dreapta. Atunci când se editează textul unui ExtendedTextBox, în momentul în care se apasă ENTER sau se părăseşte controlul respectiv, deci când se trimite comanda, Text-ul este deja modificat. Pentru a păstra într-un mod mai convenabil starea anterioară, se poate folosi proprietatea PreviousText. Cum ar putea fi proiectată comanda de modificare a textului unei celule pentru a nu fi nevoiți să folosiți această proprietate?
- Rămâne la latitudinea proiectantului să stabilească gradul de complexitate al comenzilor – dacă o comandă acționează asupra unui TextBoxGrid sau asupra unui ExtendedTextBox. De obicei, se recomandă utilizarea unor comenzi cât mai simple.

Recomandări:

- Se poate lucra cu trei clase de comenzi, pentru schimbarea textului, schimbarea formatului şi schimbarea culorii unui ExtendedTextBox;
- Comanda va primi în constructor o referință la ExtendedTextBox-ul asupra căruia se fac modificările (acesta este Receiver-ul).

Pentru a vă putea concentra exclusiv asupra aplicării şablonului, în continuare se dau câteva clase ce urmează a fi completate sau utilizate în program.

MainForm.cs

```
namespace Spreadsheet
{
   public partial class MainForm : Form
   {
      public MainForm()
      {
         InitializeComponent();
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    _grid = new TextBoxGrid(Controls, 25, 120,
      new KeyEventHandler(textBox_KeyDown),
      new EventHandler(textBox_Leave));

    _invoker = new Invoker(_grid);
  }

  TextBoxGrid _grid;
  Invoker _invoker;
  int _selected;

  private void textBox_KeyDown(object sender, KeyEventArgs e)
  {
    if (e.KeyCode == Keys.Enter)
    {
      // când se apasă ENTER într-o celulă

      ExtendedTextBox tb = (ExtendedTextBox)sender;
      string name = tb.Name;
      _selected = Convert.ToInt32(name.Substring(2));
      this.ActiveControl = _grid.GetSuccessor(_selected);
    }
  }

  private void textBox_Leave(object sender, EventArgs e)
  {
    // când o celulă nu mai este controlul selectat activ din fereastră

    Control ac = this.ActiveControl;
    ExtendedTextBox tb = (ExtendedTextBox)sender;
    string name = tb.Name;
    _selected = Convert.ToInt32(name.Substring(2));

    tb.FormatText();

    // de completat - tratarea schimbării textului

    this.ActiveControl = ac;
    UpdateUndoRedoCombos();
  }

  private void buttonColor_Click(object sender, EventArgs e)
  {
    if (colorDialog1.ShowDialog() != DialogResult.OK)
      return;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    Color c = colorDialog1.Color;
    buttonColor.ForeColor = c;
    //buttonColor.BackColor = Color.FromArgb(255 - c.R, 255 - c.G, 255 - c.B);

    // de completat
    UpdateUndoRedoCombos();
  }

  private void buttonNormal_Click(object sender, EventArgs e)
  {
    // de completat
    UpdateUndoRedoCombos();
  }

  private void buttonBold_Click(object sender, EventArgs e)
  {
    // de completat
    UpdateUndoRedoCombos();
  }

  private void buttonItalic_Click(object sender, EventArgs e)
  {
    // de completat
    UpdateUndoRedoCombos();
  }

  private void UpdateUndoRedoCombos()
  {
    comboBoxUndo.Items.Clear();
    string[] str = _invoker.UndoNames.ToArray();
    for (int i = 0; i < str.Length; i++)
      comboBoxUndo.Items.Add(str[i]);
    if (comboBoxUndo.Items.Count > 0)
    {
      comboBoxUndo.SelectedIndex = 0;
      buttonUndo.Enabled = true;
    }
    else
      buttonUndo.Enabled = false;

    comboBoxRedo.Items.Clear();
    str = _invoker.RedoNames.ToArray();
    for (int i = 0; i < str.Length; i++)
      comboBoxRedo.Items.Add(str[i]);
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
            if (comboBoxRedo.Items.Count > 0)
            {
                comboBoxRedo.SelectedIndex = 0;
                buttonRedo.Enabled = true;
            }
            else
                buttonRedo.Enabled = false;
        }

        private void buttonUndo_Click(object sender, EventArgs e)
        {
            _invoker.Undo();
            UpdateUndoRedoCombos();
        }

        private void buttonRedo_Click(object sender, EventArgs e)
        {
            _invoker.Redo();
            UpdateUndoRedoCombos();
        }

        private void buttonExit_Click(object sender, EventArgs e)
        {
            Close();
        }
    }
}
```

ExtendedTextBox.cs

```
namespace Spreadsheet
{
    class ExtendedTextBox : TextBox
    {
        private string _previousText;

        public string PreviousText
        {
            get { return _previousText; }
            set { _previousText = value; }
        }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    public ExtendedTextBox()
      : base()
    {
      _previousText = "";
    }

    public void FormatText()
    {
      double d;
      if (double.TryParse(Text, out d))
      {
        Text = d.ToString("F2");
        TextAlign = HorizontalAlignment.Right;
      }
      else
        TextAlign = HorizontalAlignment.Left;
    }
  }
}
```

TextBoxGrid.cs

```
namespace Spreadsheet
{
  class TextBoxGrid
  {
    private ExtendedTextBox[] _textBoxes;
    public const int Size = 5;

    public TextBoxGrid(Control.ControlCollection controlCollection, int left, int top,
      KeyEventHandler keyDownEvent, EventHandler leaveEvent)
    {
      _textBoxes = new ExtendedTextBox[Size * Size];
      for (int i = 0; i < Size * Size; i++)
      {
        int x = i % Size;
        int y = i / Size;

        _textBoxes[i] = new ExtendedTextBox();
        _textBoxes[i].Width = 100;
        _textBoxes[i].Height = 20;
        _textBoxes[i].Left = left + x * 100;
        _textBoxes[i].Top = top + y * 20;
        _textBoxes[i].Text = "";
        _textBoxes[i].Name = "Tb" + i;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
            _textBoxes[i].KeyDown += keyDownEvent;
            _textBoxes[i].Leave += leaveEvent;

            controlCollection.Add(_textBoxes[i]);
        }

        for (int i = 0; i < Size * Size; i++)
            _textBoxes[i].PreviousText = _textBoxes[i].Text;

        Label[] labelsX = new Label[Size];
        for (int i = 0; i < Size; i++)
        {
            labelsX[i] = new Label();
            labelsX[i].Text = ((char)(i + 'A')).ToString();
            labelsX[i].Left = left + i * 100 + 48;
            labelsX[i].Top = top - 15;
            controlCollection.Add(labelsX[i]);
        }

        Label[] labelsY = new Label[Size];
        for (int i = 0; i < Size; i++)
        {
            labelsY[i] = new Label();
            labelsY[i].Text = (i + 1).ToString();
            labelsY[i].Left = left - 15;
            labelsY[i].Height = 20;
            labelsY[i].Top = top + i * 20 + 4;
            controlCollection.Add(labelsY[i]);
        }
    }

    public void Clear()
    {
        for (int i = 0; i < Size * Size; i++)
        {
            _textBoxes[i].Clear();
            _textBoxes[i].Font = new Font(_textBoxes[i].Font, FontStyle.Regular);
            _textBoxes[i].ForeColor = Color.Black;
            _textBoxes[i].PreviousText = _textBoxes[i].Text;
        }
    }

    public ExtendedTextBox GetSuccessor(int cellNumber)
    {
        cellNumber = (cellNumber + 1) % (Size*Size);
        return _textBoxes[cellNumber];
    }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
        public string GetCoords(int cellNumber)
        {
            char tbx = (char)((cellNumber % Size) + 'A');
            int tby = cellNumber / Size + 1;
            return tbx.ToString() + tby;
        }

        public ExtendedTextBox GetCell(int cellNumber)
        {
            return _textBoxes[cellNumber];
        }
    }
}
```

ICommand.cs

```
namespace Spreadsheet
{
    interface ICommand
    {
        bool Execute();
        void Undo();
        void Redo();
    }
}
```

Invoker.cs

```
namespace Spreadsheet
{
    class Invoker
    {
        private TextBoxGrid _grid;
        private Stack<ICommand> _commands;
        private Stack<ICommand> _redoCommands;
        private Stack<string> _undoNames, _redoNames;

        public Stack<string> RedoNames
        {
            get { return _redoNames; }
        }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
        public Stack<string> UndoNames
        {
            get { return _undoNames; }
        }

        public Invoker(TextBoxGrid grid)
        {
            _grid = grid;
            _commands = new Stack<ICommand>();
            _redoCommands = new Stack<ICommand>();
            _undoNames = new Stack<string>();
            _redoNames = new Stack<string>();
        }

        public void SetAndExecute(ICommand command, string description)
        {
            // de completat
        }

        public void Undo() // (int level)
        {
            // este foarte simplă anularea mai multor niveluri de acţiuni:
            // a ultimelor "level" comenzi

            // de completat
        }

        public void Redo() // (int level)
        {
            // de completat
        }
    }
}
```

ChangeColorCommand.cs

```
namespace Spreadsheet
{
    class ChangeColorCommand : ICommand
    {
        ExtendedTextBox _textBox;
        // alte câmpuri
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    public ChangeColorCommand(ExtendedTextBox textBox, Color color)
    {
      //...
    }

    public bool Execute()
    {
      //...
      // returnează true dacă se modifică ceva în _textBox
      // returnează false dacă nu se modifică nimic
      return false;
    }

    public void Undo()
    {
      //...
    }

    public void Redo()
    {
      //...
    }
  }
}
```

ChangeFormatCommand.cs

```
namespace Spreadsheet
{
  class ChangeFormatCommand : ICommand
  {
    ExtendedTextBox _textBox;
    // alte câmpuri

    public ChangeFormatCommand(ExtendedTextBox textBox, FontStyle format)
    {
      //...
    }

    public bool Execute()
    {
      //...
      // exemplu de schimbare a corpului de literă:
      // _textBox.Font = new Font(_textBox.Font, _newStyle);
      // returnează true dacă se modifică ceva în _textBox
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
        // returnează false dacă nu se modifică nimic
        return false;
      }

      public void Undo()
      {
        //...
      }

      public void Redo()
      {
        //...
      }
    }
}
```

ChangeTextCommand.cs

```
namespace Spreadsheet
{
  class ChangeTextCommand : ICommand
  {
    ExtendedTextBox _textBox;
    // alte câmpuri

    public ChangeTextCommand(ExtendedTextBox textBox)
    {
      //...
    }

    public bool Execute()
    {
      // textul nou deja există în _textBox.Text

      //...
      // returnează true dacă se modifică ceva în _textBox
      // returnează false dacă nu se modifică nimic
      return false;
    }

    public void Undo()
    {
      //...
    }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    public void Redo()
    {
      //...
    }
  }
}
```

6.2. *Temă pentru acasă*. Extindeţi programul astfel încât să permită operaţii simple cu numerele din celule (figura 11.7).

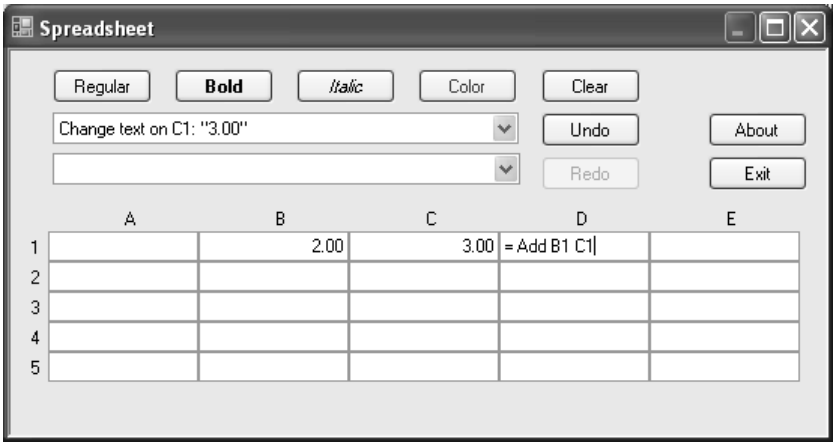


Figura 11.7. *Exemplu de rezolvare: adăugarea unor operaţii matematice simple*

La apăsarea tastei ENTER în celula D1, conţinutul său va deveni „5.00”. Operaţii permise vor fi: *Add* (adunare), *Sub* (scădere), *Mul* (înmulţire), *Div* (împărţire), cu exact 2 parametri. Se va afişa un mesaj de eroare dacă celulele trimise ca argumente nu conţin numere valide. Când cursorul revine în celula D1, deci când reîncepe editarea celulei, va apărea din nou formula. Rezultatul operaţiei apare numai când celula nu este selectată. Se vor păstra operaţiile de anulare (“undo”) şi refacere (“redo”).

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 12

Evaluarea vitezei de execuţie a unui program

1. Obiective
2. Metoda *DateTime*
3. Pointeri în C#
4. Metoda *PerformanceCounter*
5. Metoda *Stopwatch*
6. Compilarea *JIT*
7. Aplicaţii

1. Obiective

Când dorim să evaluăm performanţele unei aplicaţii software, de cele mai multe ori încercăm să-i determinăm viteza de execuţie. Există diverse strategii pentru determinarea acesteia, fie direct, prin măsurarea cât mai precisă a intervalului de timp dintre pornirea programului (sau a unei funcţii care ne interesează) şi oprirea sa, fie indirect, prin evaluarea numărului de instrucţiuni şi a naturii acestora, abordare întâlnită mai ales în studiile teoretice de determinare a complexităţii.

În acest capitol, vom descrie trei metode de măsurare directă a vitezei de execuţie a unui program.

2. Metoda *DateTime*

Prima metodă utilizează clasa DateTime. Proprietatea DateTime.Now.Ticks numără intervalele de (teoretic!) 100 de nanosecunde scurse de la ora 0 a datei de 1 ianuarie, anul 1. Pentru măsurarea timpului dintre două momente, se preia această valoare la cele două momente şi se face diferenţa. De exemplu, la pornirea unei funcţii, respectiv după terminarea execuţiei sale:

```
long startTicks = DateTime.Now.Ticks;
ApelFunctie();
double dif = (DateTime.Now.Ticks - startTicks) / 10000.0;
// dif este calculată în milisecunde
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Avantajul acestei metode este simplitatea sa din punctul de vedere al programatorului. Practic, metoda se poate folosi pentru intervale de timp mai mari, de peste 1 ms.

3. Pointeri în C#

De multe ori, funcțiile scrise în C sau C++ primesc pointeri ca parametri. Deşi în C# nu este uzual lucrul cu pointeri, acest limbaj (spre deosebire de Java) îl permite în cadrul unor construcții speciale. De exemplu, într-un DLL poate exista o funcție cu următorul prototip:

```
int NumarDivizori (int *n);
```

Importarea acestei funcții se va face astfel:

```
[DllImport("Divizibilitate", EntryPoint="NumarDivizori")]
private static unsafe extern int DllNumarDivizori(int* n);

public static unsafe int NumarDivizori(int n)
{
   int arg = n, rez = 0;

   // adresa argumentului este "fixată" în memorie, deoarece managementul automat
   // al memoriei presupune "mutarea" diferitelor zone de memorie în vederea optimizării
   fixed (int *pArg = &arg)
   {
      // apelul se face cu adresa argumentului arg
      rez = DllNumarDivizori(pArg);
   }

   return rez;
}
```

Pentru compilarea codului unsafe, trebuie prevăzută această opțiune în proprietățile proiectului din *Solution Explorer*.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

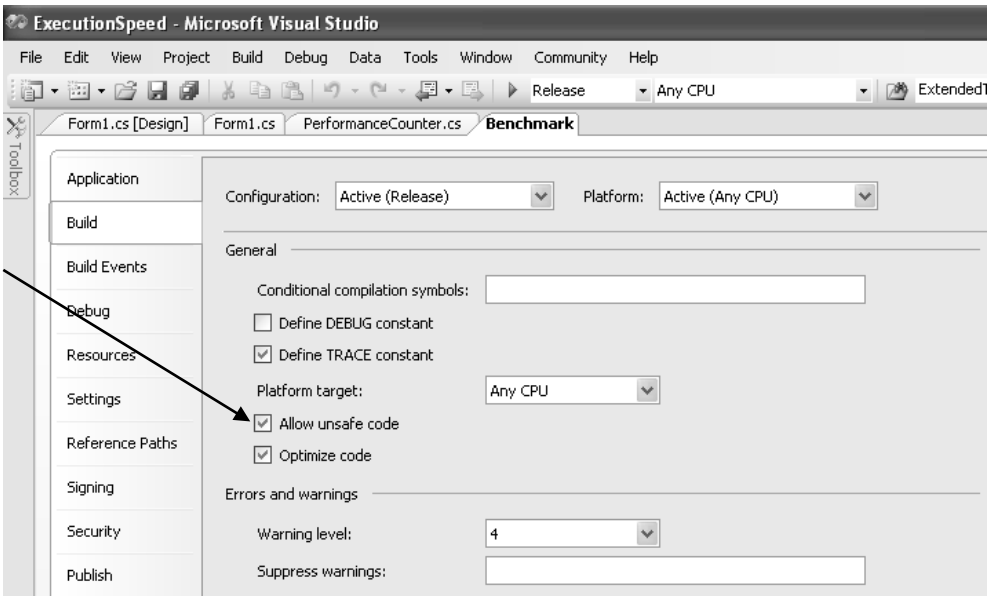


Figura 12.1. *Opţiunea pentru compilarea codului unsafe*

Să considerăm acum cazul în care funcţia C returnează un pointer. De exemplu, o funcţie pentru concatenarea a două şiruri de caractere:

```
extern "C" __declspec(dllexport) char* MyAppend(char* first, char* second)
{
   char *result = new char[strlen(first) + strlen(second)];
   strcpy(result, first);
   strcat(result, second);
   return result;
}
```

Importarea acestei funcţii se va face în felul următor:

```
[DllImport("StringDll", EntryPoint = "MyAppend")]
public static extern unsafe IntPtr StringAppend(
   [MarshalAs(UnmanagedType.LPStr)]
   string arg1,
   [MarshalAs(UnmanagedType.LPStr)]
   string arg2
);
```

Directiva MarshalAs stabileşte modul de interpretare a argumentelor şi a tipurilor de return la interfaţa dintre memoria gestionată de mediul .NET

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

(*managed*) şi cea negestionată (*unmanaged*). Interpretarea se face în momentul execuției programului. În exemplul de mai sus, funcția C primeşte un pointer la şir de caractere, care este convertit automat din tipul C# string la apel. Funcția întoarce de asemenea un pointer: IntPtr. Pentru convertirea rezultatului înapoi la string în C#, se pot folosi instrucțiuni de tipul:

```
IntPtr target = StringAppend(str1, str2);
string s = Marshal.PtrToStringAnsi(target);
```

Pentru a putea importa o funcție cu DllImport, trebuie inclus *namespace*-ul System.Runtime.InteropServices.

4. Metoda *PerformanceCounter*

Pentru intervale mai precise, se poate folosi o altă metodă care presupune apelarea directă a funcțiilor Windows de lucru cu numărătorul calculatorului:

```
[DllImport("kernel32", EntryPoint = "QueryPerformanceFrequency")]
private static unsafe extern bool QueryPerformanceFrequency(Int64* f);

[DllImport("kernel32", EntryPoint = "QueryPerformanceCounter")]
private static unsafe extern bool QueryPerformanceCounter(Int64* c);

static Int64 _t1, _t2, _htrFrecv;
static bool _htrInit;

static PerformanceCounterSpeed()
{
   // inițializarea numărătorului - o singură dată înainte de utilizarea clasei
   InitCounter();
}

private static unsafe bool InitCounter()
{
   _t1 = 0; _t2 = 0; _htrFrecv = 0; _htrInit = false;
   fixed (Int64* frecv = &_htrFrecv)
   {
      _htrInit = QueryPerformanceFrequency(frecv);
   }
   return _htrInit;
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
public unsafe void BeginTest()
{
   fixed (Int64* t1 = &_t1)
   {
      QueryPerformanceCounter(t1);
   }
}

/// <summary>
/// Returnează diferenţa în milisecunde
/// </summary>
/// <returns></returns>
public unsafe double EndTest()
{
   fixed (Int64* t2 = &_t2)
   {
      QueryPerformanceCounter(t2);
   }
   Int64 difCounts = _t2 - _t1;
   double difSeconds = (double)difCounts / (double)_htrFrecv;
   return difSeconds * 1000.0;
}
```

4.1. Metode de accelerare

O metodă mai simplă de a apela funcțiile de mai sus din *kernel32* este folosind tipul long în loc de Int64*.

Programul poate fi accelerat prin eliminarea verificărilor de securitate la apelul funcțiilor *unmanaged*, cu ajutorul atributului SuppressUnmanagedCodeSecurity:

```
[System.Security.SuppressUnmanagedCodeSecurity()]
private static extern bool QueryPerformanceCounter(out long lpPerformanceCount);
```

Când din cod *managed* se apelează funcții *unmanaged*, se realizează verificarea permisiunilor de securitate pentru a stabili dacă apelantul are drepturile necesare pentru a face apelul. Prin aplicarea explicită a acestui atribut, se elimină aceste verificări, ceea ce creşte viteza de execuție.

De asemenea, un program compilat în mod *Release* este în general mai rapid decât acelaşi cod compilat în mod *Debug*.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

5. Metoda *Stopwatch*

O metodă care combină simplitatea apelurilor oferită de clasa DateTime cu precizia funcțiilor PerformanceCounter este utilizarea clasei Stopwatch din *namespace*-ul System.Diagnostics.

Măsurarea timpului de execuție se realizează astfel:

```
long startTicks = Stopwatch.GetTimestamp();
ApelFunctie();
double dif = (Stopwatch.GetTimestamp() - _startTicks) * 1000.0 / Stopwatch.Frequency;
// dif este calculată în milisecunde
```

Indiferent de metoda de măsurare aleasă, se recomandă efectuarea mai multor experimente pentru aceeaşi funcție şi apoi calcularea statistică a valorii medii şi deviației standard.

6. Compilarea *JIT*

În mod tradițional, există două modalități principale de execuție a codului sursă pe un procesor: interpretarea şi compilarea. Codul interpretat este transformat în cod maşină în mod continuu la fiecare executare. Codul compilat static este transformat în cod maşină o singură dată, înaintea execuției.

Interpretarea necesită mai puține resurse, însă timpul de execuție este mai mare din cauza stratului software suplimentar care prelucrează instrucțiunile. Un limbaj interpretat permite unele lucruri imposibile într-un limbaj compilat, de exemplu adăugarea sau modificarea unor funcții chiar în timpul execuției. De obicei dezvoltarea aplicațiilor într-un limbaj interpretat este mai simplă deoarece nu este necesară compilarea întregului program când se doreşte testarea numai a unor secțiuni mici de cod.

Ca exemple de limbaje interpretate putem aminti versiunile inițiale de Perl, Ruby sau Matlab. Compilatoare există pentru majoritatea limbajelor de programare importante, precum C, C++, Pascal etc.

După cum am amintit şi în capitolul 1, secțiunea 3.1, compilarea „la timp" (engl. "just-in-time compilation", JIT) reprezintă o abordare hibridă, în care transformarea în cod maşină are loc continuu, ca în cazul interpretoarelor, însă codul transformat este stocat (engl. "cached") pentru a nu afecta viteza de execuție a programului. Unitatea de compilare este metoda. Unitățile sunt compilate în cod nativ doar înainte de prima utilizare.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Pe platforma .NET, toate limbajele folosesc *Common Language Runtime*, CLR, maşina virtuală responsabilă pentru gestionarea execuției programelor. CLR compilează codul intermediar CIL în instrucțiuni maşină executate de procesorul calculatorului. CLR mai asigură servicii suplimentare cum ar fi managementul memoriei, siguranța tipurilor (engl. "type safety") şi tratarea excepțiilor. Indiferent de limbajul de programare, toate programele scrise pentru platforma .NET sunt executate de către CLR.

Întrucât optimizările de bază sunt realizate în timpul compilării codului sursă, compilarea din CIL în cod maşină este mult mai rapidă. Compilatorul JIT produce cod nativ puternic optimizat pe baza datelor despre mediul de execuție, proces numit *profiling*. Acesta reprezintă capacitatea de a măsura viteza de execuție a părților unei aplicații sau a întregii aplicații, astfel încât să fie descoperite zonele care provoacă întârzieri. De exemplu, se poate descoperi care metode sunt utilizate frecvent şi acestea pot fi transformate în cod *inline*. Deşi chiar şi aceste optimizări consumă un timp suplimentar, viteza de execuție creşte la apelurile ulterioare.

Compilatorul JIT cauzează o mică întârziere la prima execuție a unei aplicații, din cauza timpului necesar pentru încărcarea şi compilarea codului intermediar. În general, cu cât mai multe optimizări realizează JIT, cu atât mai bun este codul generat, însă creşte şi întârzierea inițială. Un compilator JIT trebuie să realizeze deci un compromis între calitatea codului generat şi timpul necesar pentru compilare.

Utilitarul *Native Image Generator*, *Ngen*, poate fi o soluție pentru reducerea întârzierii inițiale. *Ngen* precompilează codul intermediar în cod maşină nativ şi de aceea nu mai este necesară compilarea în momentul execuției. Platforma .NET 2.0 precompilează toate bibliotecile DLL după instalare. Precompilarea asigură o modalitate de a îmbunătăți timpul de pornire a aplicațiilor, însă calitatea codului compilat poate fi inferioară compilării JIT, aşa cum şi codul compilat static, fără optimizări ghidate de profil, este inferioară codului compilat „la timp".

Precompilarea unui *assembly* se poate realiza cu o comandă de tipul:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\ngen.exe install nume-assembly
```

Calea către utilitar poate fi diferită, în funcție de versiunea platformei .NET utilizate. Imaginea precompilată a unui *assembly* poate fi îndepărtată din *cache* folosindu-se opțiunea uninstall.

Figura 12.2 prezintă comportamentul aceluiaşi program (un exemplu de rezolvare pentru aplicația 7.1), care măsoară viteza de execuție a unor funcții. În stânga, pentru versiunea compilată JIT, se observă că execuția

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

inițială a funcțiilor durează mai mult decât apelurile ulterioare. În dreapta, pentru versiunea precompilată cu *Ngen*, se observă că toate apelurile au un timp relativ egal de execuție.

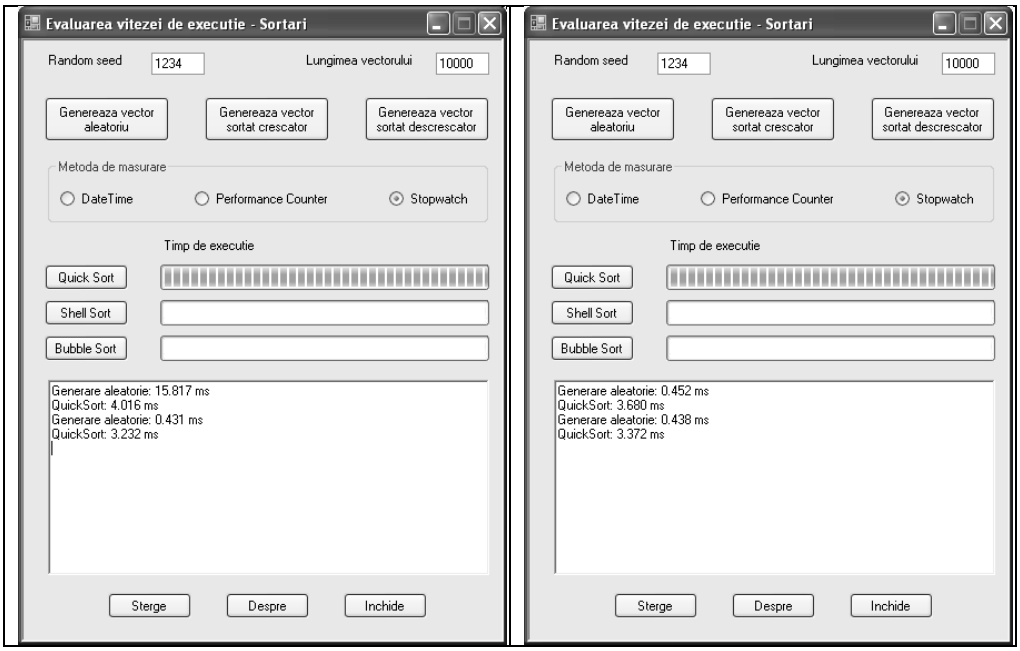


Figura 12.2. *Viteza de execuție a unor funcții în variantele: a) compilate JIT; b) precompilate*

În cazul general însă, nu se poate garanta că un cod precompilat cu *Ngen* va avea performanțe superioare față de versiunea compilată „la timp”.

Există şi metode mai complicate de a precompila *assembly*-urile prin program, însă acestea nu fac obiectul prezentului capitol.

7. Aplicaţii

7.1. Evaluați viteza de execuție a următorilor algoritmi de sortare, pentru vectori de numere reale de diferite dimensiuni: *Quick sort, Shell sort* şi *Bubble sort*.

Se va crea câte o clasă pentru fiecare din cele trei metode, care să implementeze o interfață comună:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
void BeginTest();
double EndTest(); // returnează diferenţa
```

Un exemplu de rezolvare, cu două variante de execuție, este prezentat în figura 12.3.

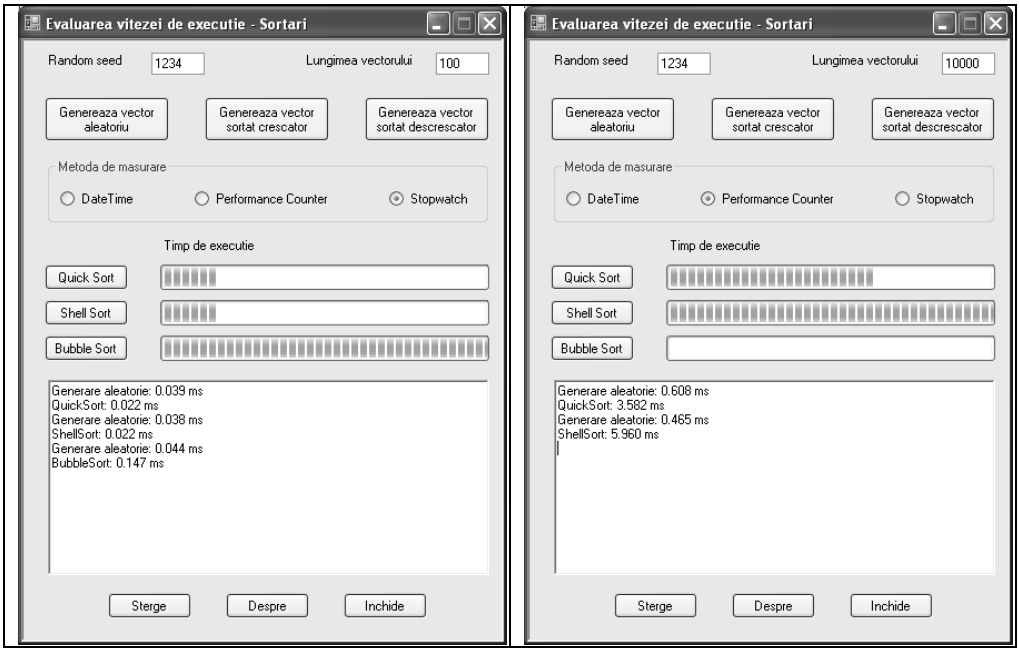


Figura 12.3. *Exemplu de rezolvare: interfaţa cu utilizatorul*

În continuare, se prezintă un schelet al clasei care implementează algoritmii de sortare considerați.

```
namespace Speed
{
   public class Sorting
   {
      public static void QuickSort(double[] vector)
      {
         DoQuickSort(vector, 0, vector.Length - 1);
      }

      private static void DoQuickSort(double[] vector, int low, int high)
      {
         if (high > low)
         {
            int k = Partition(vector, low, high); // procedura de partiţionare
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
            DoQuickSort(vector, low, k - 1);
            DoQuickSort(vector, k + 1, high);
        }
    }

    private static int Partition(double[] vector, int low, int high)
    {
        int l = low;
        int h = high;
        double x = vector[l];
        double temp;

        while (l < h)
        {
            while ((vector[l] <= x) && (l < high)) l++;
            while ((vector[h] > x) && (h >= low)) h--;

            if (l < h)
            {
                temp = vector[l];
                vector[l] = vector[h];
                vector[h] = temp;
            }
        }

        temp = vector[h];
        vector[h] = vector[low];
        vector[low] = temp;

        return h;
    }

    public static void ShellSort(double[] vector)
    {
        for (int dist = vector.Length / 2; dist > 0; dist /= 2)
            for (int i = dist; i < vector.Length; i++)
                for (int j = i - dist; j >= 0 && vector[j] > vector[j + dist]; j -= dist)
                {
                    double aux = vector[j];
                    vector[j] = vector[j + dist];
                    vector[j + dist] = aux;
                }
    }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    public static void BubbleSort(double[] vector)
    {
      // de completat cu varianta neoptimizată O(n^2)
    }
  }
}
```

Observaţii:

- Pentru a le putea compara performanţele, trebuie să ne asigurăm că toţi algoritmii sunt aplicaţi aceluiaşi vector. În cazul vectorilor generaţi aleatoriu, utili pentru studierea complexităţii medii, există două variante:
  - Să se memoreze un vector, care să fie copiat de fiecare dată înainte de aplicarea unui algoritm;
  - Să se genereze pentru fiecare algoritm *acelaşi* vector de numere aleatorii. În C#, generarea unui număr aleatoriu se face cu ajutorul unui obiect de tip Random. Acesta este de fapt un generator de numere pseudoaleatorii, în care numărul următor este generat pe baza numărului anterior, aplicând un algoritm specific. Dacă se pleacă de fiecare dată de la acelaşi număr, se va genera de fiecare dată acelaşi şir. În acest scop se poate trimite ca parametru în constructor un număr *seed*, care ajută la iniţializarea generatorului. Dacă acest parametru lipseşte (constructorul vid), se utilizează implicit timpul curent, care este de fiecare dată diferit şi deci şi şirul de numere generat va fi de asemenea diferit. Această metodă este mai eficientă deoarece elimină consumul suplimentar de memorie necesar pentru păstrarea vectorului;
- Pentru timpi de execuţie mici, este normal ca metoda *DateTime* să returneze valoarea 0;
- Nu este obligatorie afişarea grafică a comparaţiei între timpii de execuţie ai algoritmilor;
- O metodă utilă de sortare în C# este Array.Sort, care garantează complexitatea $O(n \log n)$.

7.2. *Temă pentru acasă*. Extindeţi programul prin adăugarea unui alt algoritm de sortare (la alegere) şi afişaţi grafic comparativ timpii de execuţie ai algoritmilor.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 13

Testarea unităţilor cu NUnit

1. Obiectiv
2. Testarea unităţilor
3. Utilizarea platformei *NUnit*
4. Aplicaţii

1. Obiectiv

În capitolul 13, vom descrie o metodă de testare a unităților folosind platforma dedicată *NUnit*.

2. Testarea unităţilor

Testarea unităților este o metodă de a verifica funcționalitatea corectă a unităților individuale de cod sursă. O unitate este cea mai mică parte testabilă dintr-o aplicație. Scopul testării unităților este de a izola fiecare parte a unui program şi de a arăta că părțile individuale sunt corecte. Testarea unei unități furnizează un contract scris şi strict pe care codul respectiv trebuie să îl satisfacă. Astfel, testarea unităților ajută la descoperirea din timp a problemelor în ciclul de dezvoltare.

Testarea unităților permite programatorului să refactorizeze codul mai târziu, şi asigură în continuare funcționalitatea corectă. De exemplu, testarea de regresiune presupune ca atunci când un defect este descoperit şi corectat, se înregistrează şi un test care expune defectul, astfel încât acesta să poată fi retestat după schimbările ulterioare ale programului. În mod normal sunt scrise teste pentru toate metodele astfel încât atunci când o modificare provoacă un defect, acesta să poată fi uşor identificat şi reparat. Testarea unităților ajută la eliminarea incertitudinii privind codul şi poate fi folosită într-o abordare de testare *bottom-up*. Prin testarea mai întâi a părților unui program şi apoi a sumei părților, testarea de integrare devine mult mai uşoară.

Testarea unităților asigură un fel de documentație vie a sistemului. Dezvoltatorii care vor să afle funcționalitățile oferite de o unitate şi cum este ea folosită îi pot studia testele pentru a avea o înțelegere de bază asupra

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

API-ului său. Cazurile de test pot indica folosirea potrivită sau nepotrivită a unității. Documentația narativă clasică este mai predispusă la apariția diferențelor față de implementare și deci poate deveni mai ușor depășită.

Testarea este o activitate distinctă de *depanare* (engl. "debugging"), deși granițele celor două nu sunt stricte. Testarea este procesul metodic de a demonstra *existența* defectelor din program. Depanarea este activitatea de *descoperire a cauzelor* defectelor. Testarea conduce la depanare, care conduce la reparare, care conduce la re-testare pentru a demonstra că repararea a fost corectă.

3. Utilizarea platformei *NUnit*

NUnit este o platformă *open source* pentru testarea unităților pentru aplicații .NET. Are același scop ca și *JUnit* pentru aplicații Java. Pentru a descrie modul de lucru cu *NUnit*, să considerăm o clasă pentru efectuarea unor operații artimetice elementare:

```
public class MathOps
{
   public static double Add(double x, double y)
   {
      return x + y;
   }

   public static double Sub(double x, double y)
   {
      return x - y - 1;  // rezultat incorect
   }

   public static double Mul(double x, double y)
   {
      return x * y + 0.5;     // rezultat incorect
   }

   public static double Div(double x, double y)
   {
      if (x == 0 && y == 0)
         throw new Exception("Undefined operation.");
      // if (y == 0)
      //    throw new Exception("Division by zero.");  // lipseşte excepţia
      return x / y;
   }
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    public static double Pow(double x, double y)
    {
        throw new Exception("The method or operation is not implemented.");
    }
}
```

Pentru a utiliza platforma *NUnit*, trebuie adăugată mai întâi o referinţă la *assembly*-ul nunit.framework.dll aflat în directorul NUnit 2.4.8\bin\. Pentru versiunile ulterioare este de presupus că numele acestuia se va modifica în mod corespunzător.

Pentru fiecare clasă testată, se recomandă crearea unei alte clase de test. În aceste fişiere trebuie inclus *namespace*-ului NUnit.Framework.

Pentru a indica faptul că o clasă este destinată testării cu *NUnit*, se foloseşte atributul [TestFixture].

Important! Clasele marcate [TestFixture] trebuie să fie publice.

```
[TestFixture]
public class OperatiiTest
{
    ...
}
```

În această clasă vom include metodele de testare. Aceste metode trebuie să fie obligatoriu fără parametri şi să aibă tipul de return void. O astfel de metodă este marcată cu atributul [Test]. De exemplu:

```
[Test]
public void Addition()
{
    Assert.AreEqual(3, MathOps.Add(1, 2));
}
```

Cea mai des utilizată modalitate de testare este cu clasa Assert. În metoda AreEqual primul parametru reprezintă valoarea aşteptată iar al doilea rezultatul operaţiei testate.

În legătură cu operaţiile Assert trebuie făcute câteva precizări:

- Aserţiunile nu înlocuiesc excepţiile. Excepţiile se referă la *necesităţile* codului, aserţiunile se referă la *presupunerile* codului;

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

- O aserțiune ne poate spune nu numai *ce* s-a întâmplat şi *unde* (ca o excepție), ci şi *de ce*;
- Aserțiunile pot fi considerate şi o formă de documentație, comunicând celorlalți dezvoltatori ce presupuneri s-au făcut în implementare, presupuneri de care depinde corecta funcționare a programului.

Când o aserțiune este adevărată, înseamnă că totul se desfăşoară aşa cum era prevăzut. Când este falsă, înseamnă că a apărut o eroare neaşteptată în cod. De exemplu, dacă un programator presupune că într-un fişier nu vor exista niciodată mai mult de 50000 de linii, se poate pune o aserțiune cu privire la numărul de linii din fişierul citit. Dacă programul întâlneşte un fişier mai mare, presupunerea a fost greşită şi programatorul trebuie avertizat.

Aceste metode sunt utile deci pentru depistarea erorilor semantice, aşa cum compilatorul semnalează erorile sintactice. Aserțiunile trebuie să indice erorile *programatorului*, nu ale utilizatorului.

```
[Test]
public void Subtraction()
{
   Assert.AreEqual(-1, MathOps.Sub(1, 2));
}

[Test]
public void Multiplication()
{
   Assert.AreEqual(12, MathOps.Mul(3, 4));
}

[Test]
public void Division()
{
   Assert.AreEqual(2.0 / 3.0, MathOps.Div(2, 3));
}
```

În cazul operației de împărțire, pot fi aruncate excepții dacă argumentele sunt invalide. În cazul testării, dacă argumentele sunt invalide, ne aşteptăm ca apelul să determine aruncarea unei excepții. Programul testat are un defect dacă nu se aruncă excepția. În *NUnit*, această situație este tratată cu ajutorul atributului [ExpectedException(·)], care primeşte ca parametru tipul de excepție aşteptată. În cazul nostru, avem doar excepții de

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

tipul de bază Exception. Dacă am crea clase particulare de excepție pentru fiecare situație, am putea testa că şi tipul de excepție aruncată este cel corect.

```
[Test]
[ExpectedException(typeof(Exception))]
public void DivisionExc1()
{
   MathOps.Div(0, 0);
}

[Test]
[ExpectedException(typeof(Exception))]
public void DivisionExc2()
{
   MathOps.Div(5, 0);
}
```

Când unele metode din programul testat nu sunt încă implementate, ceea ce ar determina oricum picarea testului, se poate marca o metodă de test cu atributul [Ignore], care primeşte ca parametru un mesaj de atenționare. În cazul nostru, operația Pow nu este încă implementată. Testul poate fi însă scris de la început şi când operația va fi implementată se va elimina doar atributul de ignorare din metoda de test.

```
[Test]
[Ignore("Operatia nu este inca implementata")]
public void Power()
{
   Assert.AreEqual(8, MathOps.Pow(2, 3));
}
```

În continuare, se compilează programul şi se deschide interfața vizuală a platformei *NUnit*: aplicația nunit.exe din acelaşi director NUnit 2.4.8\bin\. Se creează eventual un nou proiect şi se adaugă *assembly*-ul care trebuie testat din meniul *Project → Add Assembly…* (de exemplu Operatii.exe). Apoi se apasă butonul *Run* (figura 13.1).

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

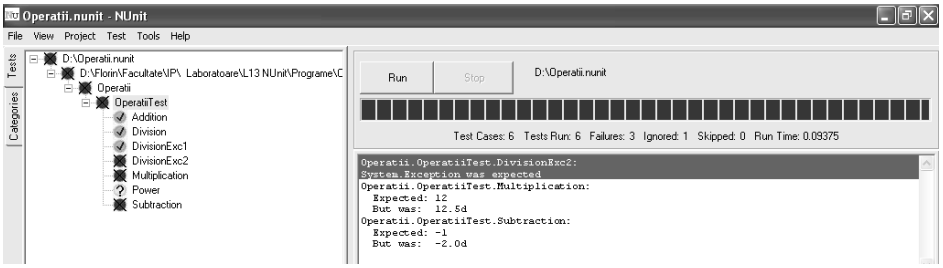


Figura 13.1. *Platforma NUnit: interfața grafică*

Pentru fiecare metodă care se testează se pot obține informații suplimentare cu click-dreapta, *Properties*. Pe lângă detalii legate de comportamentul aşteptat, sunt incluse şi evaluări ale timpului de execuție al metodelor testate.

În continuare, se face depanarea programului, se recompilează şi se rulează din nou testele în interfața *NUnit*. La apăsarea butonului *Run*, *assembly*-ul este reîncărcat, deci testele se vor face pe noua variantă a programului.

Clasa de test poate conține câmpuri proprii. Dacă este nevoie de inițializarea lor înainte de rularea celorlalte metode de test, inițializarea se realizează într-o metodă marcată cu atributul [SetUp]. De exemplu:

```
[TestFixture]
public class AccountTest
{
   Account _source;
   Account _destination;

   [SetUp]
   public void Init()
   {
      _source = new Account();
      _source.Deposit(200.00f);
      _destination = new Account();
      _destination.Deposit(150.00f);
   }

   [Test]
   public void TransferFunds()
   {
      _source.TransferFunds(_destination, 100.00f);
      Assert.AreEqual(250.00f, _destination.Balance);
      Assert.AreEqual(100.00f, _source.Balance);
   }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

NUnit permite ca o metodă să fie rulată după terminarea tuturor testelor prin marcarea ei cu atributul [TearDown]. Acestă metodă se poate folosi de exemplu pentru dezalocarea unor resurse utilizate la teste.

O clasă de test poate avea doar o singură metodă SetUp şi doar o singură metodă TearDown. Dacă sunt definite mai multe, programul va compila, dar testele nu vor rula. Metoda TearDown nu va rula dacă metoda SetUp eşuează sau aruncă o excepţie.

În concluzie, testele *NUnit* pot fi create fără a modifica programul testat şi pot fi eliminate în varianta pregătită pentru lansare. Clasa Assert este utilizată de obicei pentru compararea rezultatelor aşteptate cu cele reale. Metoda care testează egalitatea este cea mai folosită. De fiecare dată când proiectul se modifică, testele unităților trebuie rulate din nou pentru a verifica dacă funcționalitatea existentă nu a fost afectată.

4. Aplicaţii

4.1. Implementați programul cu operații aritmetice prezentat mai sus. Creați cazurile de test descrise şi rulați-le în interfața *NUnit*.

4.2. Realizați un program care calculează numărul de zile diferență între un moment referință ales arbitrar (1 ianuarie 2000) şi o dată introdusă de utilizator. Diferența de zile între oricare două date poate fi calculată prin adunarea sau scăderea rezultatelor obținute prin două apeluri succesive ale acestei funcții. Nu se vor folosi clasele specializate din C# gen DateTime.

Date de intrare: o dată calendaristică în formatul zi / lună / an.
Date de ieşire: diferența de zile (pozitivă sau negativă).

Se vor considera cazurile de test din tabelul 13.1.

Tabelul 13.1. *Cazuri de test*

Data	Rezultat corect
4.07.2003	1280
29.02.2100	dată invalidă
10.05.2000	130
5.01.2000	4
15.10.3404	513088
2.01.2009	3289
30.04.1972	-10107

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Data	Rezultat corect
15.12.1999	-17
1.01.1976	-8766
5.11.1997	-787
20.03.79	-701552
32.12.1988	dată invalidă

Diagramele UML ale celor două clase sunt prezentate în figura 13.2.

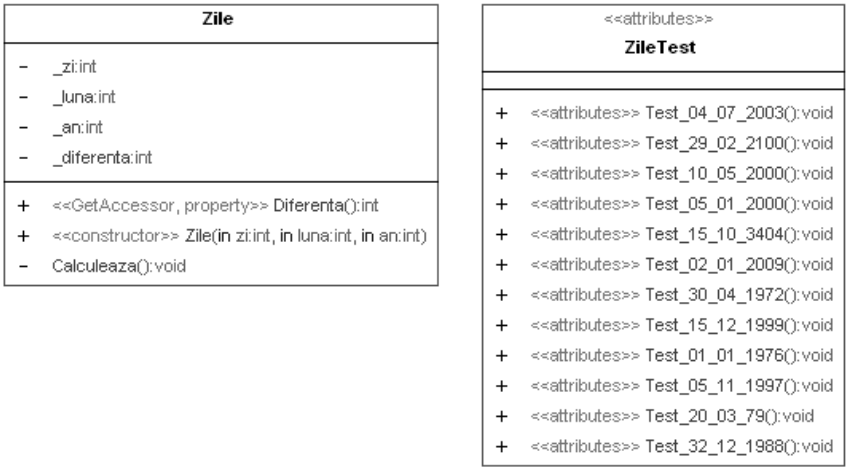


Figura 13.2. *Diagrama UML a clasei Zile și cea a clasei de test*

Indicații: Problema principală la calcularea diferenței de zile este determinarea numărului de ani bisecți. Regula pentru a determina dacă un an este bisect este următoarea: anul este divizibil cu 4, dar nu este divizibil cu 100 sau este divizibil cu 400. De exemplu, anul 1900 nu a fost bisect iar 2000 a fost an bisect pentru că este divizibil cu 400. Se poate utiliza secvența următoare de cod:

```
int[] nrzile = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

if (_an % 4 == 0 && _an % 100 != 0 || _an % 400 == 0)
   nrzile[1] = 29;
```

Proprietatea Diferenta este *readonly* (are numai accesorul get). Metoda Calculeaza aruncă o excepție dacă data este invalidă.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

O alternativă la această proiectare a clasei Zile este declararea unei metode statice de tipul:

```
public static int CalculeazaDiferenta(int zi, int luna, int an)
```

În acest fel, s-ar evita posibila aruncare a unei excepții la instanțierea unui obiect Zile, situație care nu este foarte elegantă.

Realizați *mai întâi* cazurile de test şi *apoi* implementați metodele clasei Zile, verificând la fiecare modificare rezultatele în interfața vizuală *NUnit*.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Capitolul 14

Rapoarte de testare

1. Obiective
2. Testarea unei ierarhii de clase
3. Aplicaţie

1. Obiective

Scopul capitolului 14 este prezentarea unei modalități de utilizare a platformei *NUnit* pentru testarea unei ierarhii de clase fără duplicarea codului de test şi realizarea unui raport de testare complet pentru o problemă (aparent) simplă de clasificare a triunghiurilor.

2. Testarea unei ierarhii de clase

Să considerăm un program care indică tipul unui triunghi pe baza lungimilor laturilor sale: scalen (oarecare), isoscel (două laturi egale), echilateral (toate trei laturi egale). Programul mai trebuie să verifice dacă triunghiul determinat de cele trei valori este un triunghi valid. Avem patru variante de clase: Triangle1, Triangle2, Triangle3 şi Triangle4 (figura 14.1), toate cu aceeaşi structură, fiind diferite doar implementările metodelor.

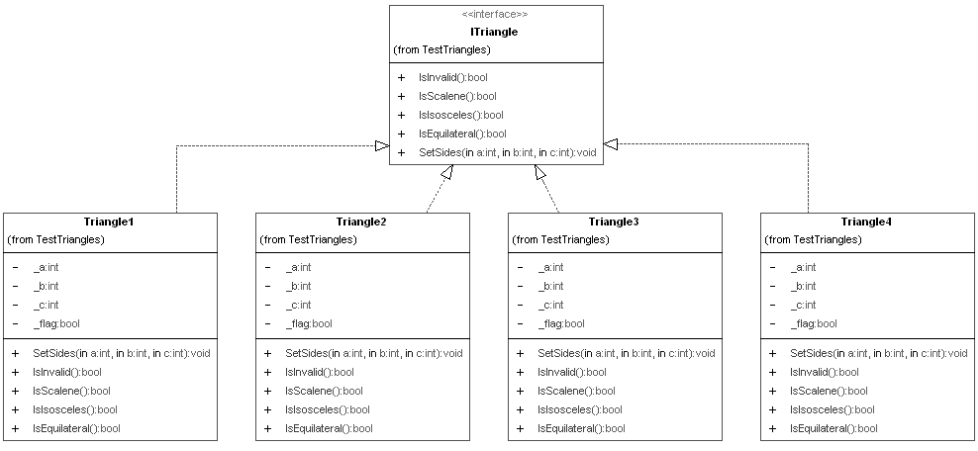


Figura 14.1. *Ierarhia claselor care trebuie testate*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Vor fi necesare mai multe teste pentru fiecare tip de triunghi, cu exact acelaşi cod, adică aceleaşi valori ale laturilor şi rezultate aşteptate. De aceea, pentru a nu repeta de patru ori acelaşi cod de test, putem scrie nişte teste generice pentru interfața ITriangle.

În plus, putem optimiza modul de verificare a rezultatelor, deoarece sunt patru metode care trebuie testate: IsScalene, IsIsosceles, IsEquilateral şi IsInvalid. Astfel, putem crea o metodă care să apeleze toate cele patru metode şi să returneze un şir de caractere care să codeze rezultatele tuturor acestor metode:

```
private string EvaluateTriangle()
{
    if (_triangle.IsInvalid())
        return "0";

    string s = "";

    if (_triangle.IsScalene())
        s += "1";
    else
        s += "0";

    if (_triangle.IsIsosceles())
        s += "1";
    else
        s += "0";

    if (_triangle.IsEquilateral())
        s += "1";
    else
        s += "0";

    return s;
}
```

Pentru un triunghi invalid, metoda va returna "0", pentru un triunghi oarecare (scalen), metoda va returna "100", pentru un triunghi isoscel va returna "010" iar pentru un triunghi echilateral "001". Din cauza faptului că metodele testate au erori, pot apărea rezultate incorecte, precum "101".

Clasa de testare generică a interfeței ITriangle va avea următoarea formă:

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
// [TestFixture]
public class TriangleTest
{
    protected ITriangle _triangle;

     [Test]
    public void ValidScalene()
    {
        // 1. Scalen valid
        _triangle.SetSides(10, 8, 5);
        Assert.AreEqual("100", EvaluateTriangle(), "10-8-5");
    }

   // ... celelalte teste
}
```

Se remarcă faptul că această clasă *nu* are atributul [TestFixture], deoarece testele vor fi realizate doar pentru tipurile concrete.

În acest scop, vom deriva patru clase din clasa TriangleTest, câte una pentru fiecare tip de triunghi. În metoda marcată cu atributul [SetUp] vom iniţializa câmpul _triangle. În continuare, platforma *NUnit* va rula testele din clasa Triangle**1**Test, moştenite din clasa de bază TriangleTest, având câmpul _triangle de tipul triunghiului concret Triangle1.

```
[TestFixture]
public class Triangle1Test : TriangleTest
{
    [SetUp]
    public void Init()
    {
        _triangle = new Triangle1();
    }
}
```

În mod similar se procedează şi pentru celelalte tipuri concrete.

Rezultatele testării vor apărea ca în figura 14.2.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

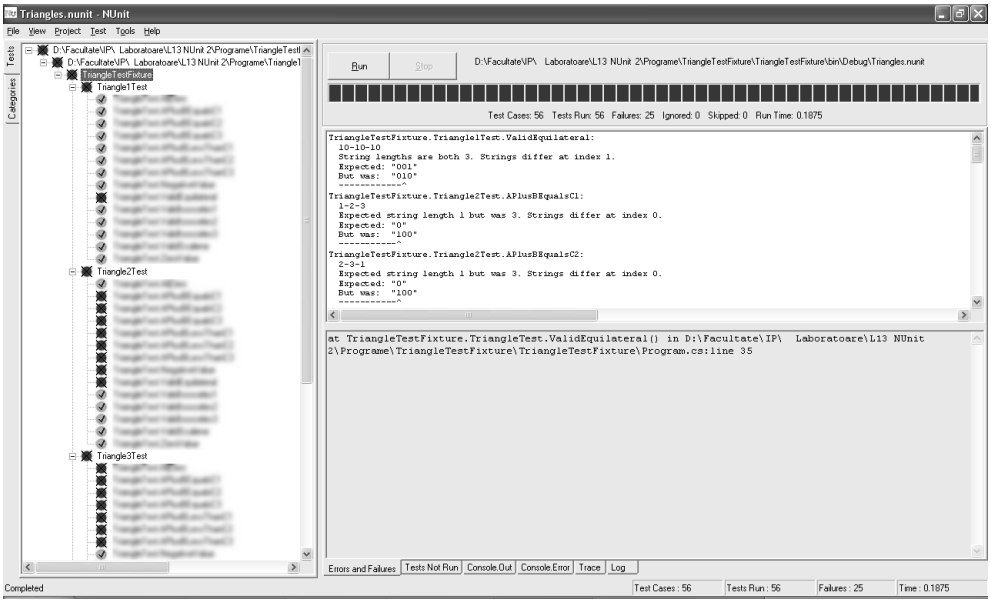


Figura 14.2. *Rezultatele testării cu NUnit*

3. Aplicaţie

3.1. Se doreşte testarea „cutie neagră” (engl. “black box”) a claselor de triunghiuri descrise anterior. Folosiți platforma *NUnit* pentru a testa toate cazurile de eroare care pot apărea. Vom considera că dacă triunghiul este valid, o singură metodă trebuie să fie activă la un moment dat. De exemplu, considerăm că un triunghi echilateral nu este în acelaşi timp şi isoscel.

Implementările claselor din ierarhia de triunghiuri sunt prezentate în continuare.

```
namespace TestTriangles
{
   public interface ITriangle
   {
      bool IsInvalid();
      bool IsScalene();
      bool IsIsosceles();
      bool IsEquilateral();
      void SetSides(int a, int b, int c);
   }
}
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
namespace TestTriangles
{
   public class Triangle1 : ITriangle
   {
      int _a;
      int _b;
      int _c;
      bool _flag;

      public void SetSides(int a, int b, int c)
      {
         _a = a;
         _b = b;
         _c = c;
         int semiperim = (a + b + c) / 2;
         _flag = a == 0 || b == 0 || c == 0 || a < 0 || b < 0 || c < 0 || semiperim <= a ||
            semiperim <= b || semiperim <= c;
      }

      public bool IsInvalid()
      {
         return _flag;
      }

      public bool IsScalene()
      {
         return _a != _b && _a != _c && _b != _c;
      }

      public bool IsIsosceles()
      {
         return _a == _b || _a == _c || _b == _c;
      }

      public bool IsEquilateral()
      {
         return _a == _b && _b == _c && !IsIsosceles();
      }
   }
}
```

```
namespace TestTriangles
{
   public class Triangle2 : ITriangle
   {
      int _a;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
      int _b;
      int _c;
      bool _flag;

      public void SetSides(int a, int b, int c)
      {
         _a = a;
         _b = b;
         _c = c;
         int semiperim = (a + b + c) / 2;
         _flag = a == 0 || b == 0 || c == 0;
      }

      public bool IsInvalid()
      {
         return _flag;
      }

      public bool IsScalene()
      {
         return _a != _b && _a != _c && _b != _c;
      }

      public bool IsIsosceles()
      {
         return _a == _b || _a == _c || _b == _c;
      }

      public bool IsEquilateral()
      {
         return _a == _b && _b == _c;
      }
   }
}
```

```
namespace TestTriangles
{
   public class Triangle3 : ITriangle
   {
      int _a;
      int _b;
      int _c;
      bool _flag;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
    public void SetSides(int a, int b, int c)
    {
      _a = a;
      _b = b;
      _c = c;
      int semiperim = (a + b + c) / 2;
      _flag = a < 0 || b < 0 || c < 0;
    }

    public bool IsInvalid()
    {
      return _flag;
    }

    public bool IsScalene()
    {
      return _a != _b && _a != _c && _b != _c;
    }

    public bool IsIsosceles()
    {
      return _a == _b || _a == _c || _b == _c;
    }

    public bool IsEquilateral()
    {
      return _a == _b && _b == _c;
    }
  }
}
```

```
namespace TestTriangles
{
  public class Triangle4 : ITriangle
  {
    int _a;
    int _b;
    int _c;
    bool _flag;

    public void SetSides(int a, int b, int c)
    {
      _a = a;
      _b = b;
      _c = c;
      int semiperim = (a + b + c) / 2;
```


Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

```
        _flag = a == 0 || b == 0 || c == 0 || a < 0 || b < 0 || c < 0;
    }

    public bool IsInvalid()
    {
      return _flag;
    }

    public bool IsScalene()
    {
      return _a != _b && _a != _c && _b != _c;
    }

    public bool IsIsosceles()
    {
      return _a == _b || _a == _c || _b == _c;
    }

    public bool IsEquilateral()
    {
      return _a == _b && _b == _c;
    }
  }
}
```

Creați un raport de testare pentru fiecare clasă. Raportul va avea structura din tabelul 14.1.

Tabelul 14.1. *Raport de testare*

Nr. test	Valori laturi	Răspuns corect	Răspuns program	Observații
1	3 5 5	Isoscel	Scalen, Isoscel	incorect
2	3 3 3	Echilateral	Echilateral	corect
...				

Scopul este să descoperiți cât mai multe cazuri de test, cu valori atât corecte cât şi incorecte. O listă cuprinzătoare a cazurilor de test este dată în figura 14.3. Consultați-o doar după ce ați încercat propriile cazuri.

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

1. Triunghi scalen valid (cazuri precum 1,2,3 și 2,5,10 nu garantează un răspuns „da”, deoarece triunghiul este invalid);
2. Triunghi echilateral valid;
3. Triunghi isoscel valid (un caz precum 2,2,4 nu contează, deoarece nu este un triunghi valid);
4. Cel puțin trei cazuri pentru punctul anterior, în care se încearcă toate trei permutări de laturi egale (de exemplu 3,3,4 – 3,4,3 – 4,3,3);
5. Cazuri precum 2,2,3 – 2,3,2 – 3,2,2 (diferență par-impar față de punctul anterior);
6. O latură cu valoarea 0;
7. O latură cu valoare negativă;
8. Trei întregi mai mari decât 0 astfel încât suma a două numere este egală cu al treilea, astfel încât, dacă programul consideră că 1,2,3 este un triunghi scalen (oarecare), există o eroare;
9. Cel puțin trei cazuri pentru punctul anterior, cu cele trei permutări ale laturii „lungi” (de exemplu 1,2,3 – 1,3,2 – 3,1,2);
10. Trei întregi mai mari decât 0 astfel încât suma a două numere este mai mare decât al treilea (de exemplu 1,2,4 sau 12,15,30);
11. Cel puțin trei cazuri pentru punctul anterior, cu cele trei permutări ale laturii „lungi” (de exemplu 1,2,4 – 1,4,2 – 4,1,2);
12. Toate cele trei laturi sunt 0.

Dacă nu ar fi fost o implementare C#, s-ar mai fi putut testa răspunsul programului la argumente neîntregi sau apelul cu un număr diferit de argumente, de exemplu 2 în loc de 3.

Figura 14.3. *Exemplu de rezolvare: cazuri de test*

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

Referințe

[1] Albahari, J. (2008). *C# Concepts: Value vs Reference Types*, http://www.albahari.com/valuevsreftypes.aspx

[2] Altova Umodel (2008). *UModel tutorial*, http://www.altova.com/manual2008/UModel/umodelprofessional

[3] Ariadne Training (2008). *UML Applied*, 2nd edition, http://ariadnetraining.com

[4] Becker, K. (2008). *Using the Model View Controller Pattern in C# ASP .NET Web Applications*, http://www.primaryobjects.com/CMS/Article82.aspx

[5] Data & Object Factory (2002). *Software Design Patterns*, http://www.dofactory.com/Patterns/Patterns.aspx

[6] Fowler, M. (2006). *Passive View*, http://martinfowler.com/eaaDev/PassiveScreen.html

[7] Fowler, M. (2006). *Supervising Controller*, http://martinfowler.com/eaaDev/SupervisingPresenter.html

[8] Freeman, E., Freeman, E., Bates, B., Sierra, K. (2004). *Head First Design Patterns*, O'Reilly Media, Inc.

[9] Gamma, E., Helm, R., Johnson, R., Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional

[10] Goodliffe, P. (2006). *Code Craft: The Practice of Writing Excellent Code*, No Starch Press

[11] Hilyard, J., Teilhet, S. (2006). *C# Cookbook*, 2nd Edition, O'Reilly

[12] Liptchinsky, V. (2008). *Pre-compile (pre-JIT) your assembly on the fly, or trigger JIT compilation ahead-of-time*, http://www.codeproject.com/Articles/31316/Pre-compile-pre-JIT-your-assembly-on-the-fly-or-tr

[13] Microsoft MSDN Library (2008-2011). Paginile despre clase, structuri, enumerații, Model-View-Controller, Model-View-Presenter, clase parțiale, proprietăți, clase abstracte, interfețe, polimorfism, http://msdn.microsoft.com/en-us/library/default.aspx

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com

[14] MSDN Forum (2008). *Faster Deep Cloning*, http://social.msdn.microsoft.com/Forums/en-US/csharpgeneral/thread/5e13351f-abdb-42d4-93f0-64bff65b8d74

[15] MVCSharp Org. (2011). *MVC# Overview*, http://www.mvcsharp.org/Overview/Default.aspx

[16] Myers, G. J., Sandler, C., Badgett, T., Thomas, T. M. (2004). *The Art of Software Testing*, 2nd edition, Wiley

[17] Obviex (2010). *How To: Encrypt and Decrypt Data Using a Symmetric (Rijndael) Key (C#/VB.NET)*, http://www.obviex.com/samples/Encryption.aspx

[18] SmartForce Ireland Ltd & Classic Systems Solutions (1999). *GUI Design Fundamentals*

[19] SpiderWorks Technologies (2011). *C# Coding Standards and Best Programming Practices*, http://www.dotnetspider.com/tutorials/BestPractices.aspx

[20] Wikipedia, The Free Encyclopedia (2008). Paginile despre CIL/MSIL, Reflector .NET, DLL, DLL hell, Unit testing, Model-View-Controller, Just-in-time_compilation, http://en.wikipedia.org

[21] Ziff Davis, Inc. (2009). *Adopting C#, and Eluding DLL Hell*, http://www.extremetech.com/article2/0,2845,1153065,00.asp

Florin Leon (2012). Aplicatii de ingineria programarii in C#, Tehnopress, Iasi, ISBN 978-973-702-909-6
http://florinleon.byethost24.com